Programming languages — C

Abstract

(This cover sheet to be replaced by ISO.)

This document specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems.

Clauses are included that detail the C language itself and the contents of the C language execution library. Annexes summarize aspects of both of them, and enumerate factors that influence the portability of C programs.

Although this document is intended to guide knowledgeable C language programmers as well as implementors of C language translation systems, the document itself is not designed to serve as a tutorial.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

This document marks changes from version C17 to version C2x. They are indicated by striking out text that has been deleted and underlining text that has been added. Pages that contain changes are marked with C17..C2x and are listed under C2x CHANGES in the index.

The following documents have been applied to this draft:

- DR 476 volatile semantics for lvalues
- DR 488 `c16rtomb()` on wide characters encoded as multiple `char16_t`
- DR 494 Part 1: Alignment specifier expression evaluation
- DR 496 `offsetof` and subobjects (with editorial modification)
- DR 497 "white-space character" defined in two places
- DR 499 Anonymous structure in union behavior
- DR 500 Ambiguous specification for `FLT_EVAL_METHOD`
- DR 501 make `DECIMAL_DIG` obsolescent
- FP DR 20 changes for obsolescing `DECIMAL_DIG`
- FP DR 21 `printf` of one-digit character string
- FP DR 23 `llquantexp` invalid case
- FP DR 24 `remainder` NaN case
- FP DR 25 `totalorder` parameters
- N2124 and N2319 rounding direction macro `FE_TONEARESTFROMZERO`
- N2186 Alternative to N2166
- N2212 type generic `cbrt` (with editorial changes)
N2260 Clarifying the `restrict` keyword v2
N2265 Harmonizing `static_assert` with C++
N2267 `nodiscard` attribute
N2270 `maybe_unused` attribute
N2271 CR for `pow` divide-by-zero case
N2293 Alignment requirements for memory management functions
N2314 TS 18661-1 plus CR/DRs for C2X
N2322 preprocessor line numbers unspecified
N2325 `DBL_NORM_MAX` etc
N2326 floating-point zero and other normalization
N2334 `deprecated` attribute
N2335 attributes
N2337 `strftime`, with ‘b’ and ‘B’ swapped
N2338 error indicator for encoding errors in `fgetwc`
N2341 TS 18661-2 plus CR/DRs for C2X
N2345 editors, resolve ambiguity of a semicolon
N2349 the `memccpy` function
N2350 defining new types in `offsetof`
N2353 the `strdup` and `strndup` functions
N2356 update for payload functions
N2358 no internal state for `mblen`
N2359 part 2 (remove `WANT` macros from numbered clauses) and part 3 (version macros for changed library clauses)
N2401 TS 18661-4a for C2X
N2408 The `fallthrough` attribute
N2417 Section 6: Add time conversion functions that are relatively thread-safe
N2418 Adding the `u8` character prefix
N2432 Remove support for function definitions with identifier lists

In addition to these, the document has undergone some editorial changes, namely

— The synopsis lists in Annex B are now generated automatically and classified according to the feature test or `WANT` macros that are required to make them available.
— Addition of a new non-normative clause J.6 to Annex J that categorizes identifiers used by this document.
— Renaming of the syntax term “`struct declaration`”, “`struct declaration list`”, “`struct declarator`”, and “`struct declarator list`” to the more appropriate “`member declaration`”, “`member declaration list`”, “`member declarator`” and “`member declarator list`”, respectively.
Contents

Foreword xiii
Introduction xv
1 Scope 1
2 Normative references 2
3 Terms, definitions, and symbols 3
4 Conformance 8
5 Environment 9
   5.1 Conceptual models 9
      5.1.1 Translation environment 9
      5.1.2 Execution environments 10
   5.2 Environmental considerations 17
      5.2.1 Character sets 17
      5.2.2 Character display semantics 19
      5.2.3 Signals and interrupts 19
      5.2.4 Environmental limits 19
6 Language 33
   6.1 Notation 33
   6.2 Concepts 33
      6.2.1 Scopes of identifiers 33
      6.2.2 Linkages of identifiers 34
      6.2.3 Name spaces of identifiers 34
      6.2.4 Storage durations of objects 35
      6.2.5 Types 36
      6.2.6 Representations of types 39
      6.2.7 Compatible type and composite type 40
      6.2.8 Alignment of objects 41
   6.3 Conversions 42
      6.3.1 Arithmetic operands 42
      6.3.2 Other operands 45
   6.4 Lexical elements 48
      6.4.1 Keywords 49
      6.4.2 Identifiers 49
6.4.3 Universal character names ........................................... 51
6.4.4 Constants ................................................................. 52
6.4.5 String literals ............................................................ 59
6.4.6 Punctuators ............................................................... 61
6.4.7 Header names ............................................................ 61
6.4.8 Preprocessing numbers ................................................ 62
6.4.9 Comments ................................................................. 62

6.5 Expressions ................................................................. 64
   6.5.1 Primary expressions ................................................ 65
   6.5.2 Postfix operators .................................................... 66
   6.5.3 Unary operators ..................................................... 72
   6.5.4 Cast operators ....................................................... 75
   6.5.5 Multiplicative operators ......................................... 75
   6.5.6 Additive operators ................................................ 76
   6.5.7 Bitwise shift operators .......................................... 77
   6.5.8 Relational operators .............................................. 77
   6.5.9 Equality operators ................................................. 78
   6.5.10 Bitwise AND operator .......................................... 79
   6.5.11 Bitwise exclusive OR operator ............................... 79
   6.5.12 Bitwise inclusive OR operator ............................... 80
   6.5.13 Logical AND operator .......................................... 80
   6.5.14 Logical OR operator ............................................ 80
   6.5.15 Conditional operator ........................................... 81
   6.5.16 Assignment operators .......................................... 82
   6.5.17 Comma operator .................................................. 84

6.6 Constant expressions .................................................. 85

6.7 Declarations ............................................................. 87
   6.7.1 Storage-class specifiers ...................................... 88
   6.7.2 Type specifiers .................................................... 89
   6.7.3 Type qualifiers ................................................... 97
   6.7.4 Function specifiers .............................................. 102
   6.7.5 Alignment specifier .............................................. 103
   6.7.6 Declarators ......................................................... 104
   6.7.7 Type names .......................................................... 110
   6.7.8 Type definitions .................................................. 111
   6.7.9 Initialization ....................................................... 112
   6.7.10 Static assertions ................................................. 117
   6.7.11 Attributes ......................................................... 117

6.8 Statements and blocks ................................................ 123
   6.8.1 Labeled statements .............................................. 123
7.3.4 The CX_LIMITED_RANGE pragma ................................................. 158
7.3.5 Trigonometric functions ......................................................... 158
7.3.6 Hyperbolic functions ............................................................ 160
7.3.7 Exponential and logarithmic functions ....................................... 161
7.3.8 Power and absolute-value functions ......................................... 162
7.3.9 Manipulation functions ......................................................... 163
7.4 Character handling <ctype.h> .................................................... 166
7.4.1 Character classification functions .......................................... 166
7.4.2 Character case mapping functions .......................................... 168
7.5 Errors <errno.h> ................................................................. 170
7.6 Floating-point environment <fenv.h> ......................................... 171
7.6.1 The FENV_ACCESS pragma ................................................... 173
7.6.2 The FENV_ROUND pragma .................................................... 174
7.6.3 The FENV_DEC_ROUND pragma ............................................. 175
7.6.4 Floating-point exceptions ................................................... 176
7.6.5 Rounding and other control modes ......................................... 179
7.6.6 Environment .................................................................. 181
7.7 Characteristics of floating types <float.h> .................................... 183
7.8 Format conversion of integer types <inttypes.h> ............................... 184
7.8.1 Macros for format specifiers ................................................. 184
7.8.2 Functions for greatest-width integer types .................................. 185
7.9 Alternative spellings <iso646.h> ................................................ 187
7.10 Sizes of integer types <limits.h> ............................................... 188
7.11 Localization <locale.h> ........................................................... 189
7.11.1 Locale control ................................................................. 189
7.11.2 Numeric formatting convention inquiry ................................... 190
7.12 Mathematics <math.h> ........................................................... 195
7.12.1 Treatment of error conditions .............................................. 198
7.12.2 The FP_CONTRACT pragma .................................................. 199
7.12.3 Classification macros .......................................................... 199
7.12.4 Trigonometric functions ...................................................... 202
7.12.5 Hyperbolic functions .......................................................... 207
7.12.6 Exponential and logarithmic functions ...................................... 209
7.12.7 Power and absolute-value functions ....................................... 216
7.12.8 Error and gamma functions ................................................ 220
7.12.9 Nearest integer functions ..................................................... 221
7.12.10 Remainder functions ........................................................ 225
7.12.11 Manipulation functions ....................................................... 226
7.12.12 Maximum, minimum, and positive difference functions ............. 229
7.12.13 Floating multiply-add ......................................................... 231
7.12.14 Functions that round result to narrower type ........................................... 232
7.12.15 Quantum and quantum exponent functions .............................................. 234
7.12.16 Decimal re-encoding functions ............................................................... 235
7.12.17 Comparison macros ................................................................................... 237
7.13 Nonlocal jumps <setjmp.h> ............................................................................ 240
  7.13.1 Save calling environment ........................................................................... 240
  7.13.2 Restore calling environment ...................................................................... 240
7.14 Signal handling <signal.h> ............................................................................. 242
  7.14.1 Specify signal handling .............................................................................. 242
  7.14.2 Send signal ................................................................................................ 243
7.15 Alignment <stdalign.h> ................................................................................. 245
7.16 Variable arguments <stdarg.h> ................................................................. 246
  7.16.1 Variable argument list access macros ......................................................... 246
7.17 Atomics <stdatomic.h> .................................................................................. 249
  7.17.1 Introduction ............................................................................................... 249
  7.17.2 Initialization .............................................................................................. 250
  7.17.3 Order and consistency ................................................................................ 250
  7.17.4 Fences ........................................................................................................ 253
  7.17.5 Lock-free property ...................................................................................... 254
  7.17.6 Atomic integer types ................................................................................... 254
  7.17.7 Operations on atomic types ....................................................................... 255
  7.17.8 Atomic flag type and operations ............................................................... 257
7.18 Boolean type and values <stdbool.h> ........................................................... 259
7.19 Common definitions <stddef.h> ................................................................. 260
7.20 Integer types <stdint.h> ................................................................................ 261
  7.20.1 Integer types .............................................................................................. 261
  7.20.2 Limits of specified-width integer types ....................................................... 262
  7.20.3 Limits of other integer types ...................................................................... 265
  7.20.4 Macros for integer constants ..................................................................... 266
7.21 Input/output <stdio.h> .................................................................................. 267
  7.21.1 Introduction ............................................................................................... 267
  7.21.2 Streams .................................................................................................... 268
  7.21.3 Files ........................................................................................................... 269
  7.21.4 Operations on files ..................................................................................... 271
  7.21.5 File access functions ................................................................................ 272
  7.21.6 Formatted input/output functions ............................................................. 275
  7.21.7 Character input/output functions ............................................................. 292
  7.21.8 Direct input/output functions ................................................................... 295
  7.21.9 File positioning functions ....................................................................... 296
  7.21.10 Error-handling functions ..................................................................... 298
7.22 General utilities `<stdlib.h>` .................................................. 300
  7.22.1 Numeric conversion functions ................................................. 300
  7.22.2 Pseudo-random sequence generation functions ............................ 306
  7.22.3 Memory management functions ............................................... 307
  7.22.4 Communication with the environment ...................................... 309
  7.22.5 Searching and sorting utilities .......................................... 312
  7.22.6 Integer arithmetic functions .............................................. 314
  7.22.7 Multibyte/wide character conversion functions ........................ 315
  7.22.8 Multibyte/wide string conversion functions ............................ 316

7.23 `__Noreturn` `<stdnoreturn.h>` .................................................. 318

7.24 String handling `<string.h>` .................................................. 319
  7.24.1 String function conventions .............................................. 319
  7.24.2 Copying functions .......................................................... 319
  7.24.3 Concatenation functions .................................................. 320
  7.24.4 Comparison functions ..................................................... 321
  7.24.5 Search functions .......................................................... 322
  7.24.6 Miscellaneous functions .................................................. 325

7.25 Type-generic math `<tgmath.h>` .............................................. 327

7.26 Threads `<threads.h>` ............................................................ 332
  7.26.1 Introduction ................................................................. 332
  7.26.2 Initialization functions ................................................... 333
  7.26.3 Condition variable functions ............................................ 333
  7.26.4 Mutex functions ............................................................ 335
  7.26.5 Thread functions .......................................................... 337
  7.26.6 Thread-specific storage functions .................................... 339

7.27 Date and time `<time.h>` ...................................................... 342
  7.27.1 Components of time ......................................................... 342
  7.27.2 Time manipulation functions ............................................ 343
  7.27.3 Time conversion functions .............................................. 345

7.28 Unicode utilities `<uchar.h>` ................................................... 350
  7.28.1 Restartable multibyte/wide character conversion functions ............ 350

7.29 Extended multibyte and wide character utilities `<wchar.h>` ............... 353
  7.29.1 Introduction ............................................................... 353
  7.29.2 Formatted wide character input/output functions ...................... 354
  7.29.3 Wide character input/output functions ................................ 366
  7.29.4 General wide string utilities ......................................... 370
    7.29.4.1 Wide string numeric conversion functions ....................... 370
    7.29.4.2 Wide string copying functions .................................. 374
    7.29.4.3 Wide string concatenation functions ............................ 375
    7.29.4.4 Wide string comparison functions ............................... 375
7.29.4.5 Wide string search functions ........................................ 377
7.29.4.6 Miscellaneous functions ........................................... 380
7.29.5 Wide character time conversion functions ............................... 380
7.29.6 Extended multibyte/wide character conversion utilities ............... 381
7.29.6.1 Single-byte/wide character conversion functions .................. 381
7.29.6.2 Conversion state functions ....................................... 381
7.29.6.3 Restartable multibyte/wide character conversion functions ........ 382
7.29.6.4 Restartable multibyte/wide string conversion functions .......... 383

7.30 Wide character classification and mapping utilities <wctype.h> ........... 386
7.30.1 Introduction .................................................................. 386
7.30.2 Wide character classification utilities .................................. 386
7.30.2.1 Wide character classification functions ......................... 386
7.30.2.2 Extensible wide character classification functions ......... 389
7.30.3 Wide character case mapping utilities .................................. 390
7.30.3.1 Wide character case mapping functions ......................... 390
7.30.3.2 Extensible wide character case mapping functions .......... 390

7.31 Future library directions .......................................................... 392
7.31.1 Complex arithmetic <complex.h> ...................................... 392
7.31.2 Character handling <ctype.h> .......................................... 392
7.31.3 Errors <errno.h> .......................................................... 392
7.31.4 Floating-point environment <fenv.h> .................................. 392
7.31.5 Characteristics of floating types <float.h> ............................ 392
7.31.6 Format conversion of integer types <inttypes.h> ....................... 392
7.31.7 Localization <locale.h> .................................................. 392
7.31.8 Mathematics <math.h> .................................................... 392
7.31.9 Signal handling <signal.h> ............................................. 393
7.31.10 Atomics <stdatomic.h> ................................................. 393
7.31.11 Boolean type and values <stdbool.h> ................................ 393
7.31.12 Integer types <stdint.h> ............................................... 393
7.31.13 Input/output <stdio.h> ................................................ 393
7.31.14 General utilities <stdlib.h> .......................................... 393
7.31.15 String handling <string.h> ............................................ 393
7.31.16 Date and time <time.h> .............................................. 393
7.31.17 Threads <threads.h> .................................................. 394
7.31.18 Extended multibyte and wide character utilities <wchar.h> ........ 394
7.31.19 Wide character classification and mapping utilities <wctype.h> ... 394

Annex A (informative) Language syntax summary ................................. 395
Annex B (informative) Library summary ............................................. 410
<table>
<thead>
<tr>
<th>Annex C (informative) Sequence points</th>
<th>436</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annex D (normative) Universal character names for identifiers</td>
<td>437</td>
</tr>
<tr>
<td>Annex E (informative) Implementation limits</td>
<td>438</td>
</tr>
<tr>
<td>Annex F (normative) IEC 60559 floating-point arithmetic</td>
<td>440</td>
</tr>
<tr>
<td>Annex G (normative) IEC 60559-compatible complex arithmetic</td>
<td>471</td>
</tr>
<tr>
<td>Annex H (informative) Language independent arithmetic</td>
<td>482</td>
</tr>
<tr>
<td>Annex I (informative) Common warnings</td>
<td>486</td>
</tr>
<tr>
<td>Annex J (informative) Portability issues</td>
<td>487</td>
</tr>
<tr>
<td>Annex K (normative) Bounds-checking interfaces</td>
<td>522</td>
</tr>
<tr>
<td>Annex L (normative) Analyzability</td>
<td>570</td>
</tr>
<tr>
<td>Annex M (informative) Change History</td>
<td>572</td>
</tr>
<tr>
<td>Bibliography</td>
<td>575</td>
</tr>
<tr>
<td>Index</td>
<td>576</td>
</tr>
</tbody>
</table>
Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO’s adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces.

This fourth edition cancels and replaces the third edition, ISO/IEC 9899:2011, which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 9899:2011/Cor 1:2012.

There are no major changes in this edition, only technical corrections and clarifications. Major changes from the previous edition include:

— added a one-argument version of _Static_assert
— support for function definitions with identifier lists has been removed
— harmonization with ISO/IEC 9945 (POSIX):
  • extended month name formats for strftime
  • integration of functions: asctime_r, ctime_r, gmtime_r, localtime_r, memccpy, strdup, strndup
— harmonization with floating point standard IEC 60559:
  • integration of binary floating-point technical specification TS 18661-1
  • integration of decimal floating-point technical specification TS 18661-2
  • integration of decimal floating-point technical specification TS 18661-4a
— the macro DECIMAL_DIG is declared obsolescent
— added version test macros to certain library headers
— added the attributes feature
— added deprecated, fallthrough, maybe_unused, and nodiscard attributes
— added the u8 character prefix

8 A complete change history can be found in Annex M.
Introduction

With the introduction of new devices and extended character sets, new features could be added to this document. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, could conflict with future additions.

Certain features are obsolescent, which means that they could be considered for withdrawal in future revisions of this document. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.31]) is discouraged.

This document is divided into four major subdivisions:

- preliminary elements (Clauses 1–4);
- the characteristics of environments that translate and execute C programs (Clause 5);
- the language syntax, constraints, and semantics (Clause 6);
- the library facilities (Clause 7).

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this document. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes define optional features, provide additional information and summarize the information contained in this document. A bibliography lists documents that were referred to during the preparation of this document.

The language clause (Clause 6) is derived from “The C Reference Manual”.

The library clause (Clause 7) is based on the 1984 /usr/group Standard.

The Working Group responsible for this document (WG 14) maintains a site on the World Wide Web at http://www.open-std.org/JTC1/SC22/WG14/ containing ancillary information that may be of interest to some readers such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.
Programming languages — C

1. Scope

This document specifies the form and establishes the interpretation of programs written in the C programming language. It specifies

— the representation of C programs;
— the syntax and constraints of the C language;
— the semantic rules for interpreting C programs;
— the representation of input data to be processed by C programs;
— the representation of output data produced by C programs;
— the restrictions and limits imposed by a conforming implementation of C.

This document does not specify

— the mechanism by which C programs are transformed for use by a data-processing system;
— the mechanism by which C programs are invoked for use by a data-processing system;
— the mechanism by which input data are transformed for use by a C program;
— the mechanism by which output data are transformed after being produced by a C program;
— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

---

1) This document is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and programmers. Annex J gives an overview of portability issues that a C program might encounter.
2. Normative references

1. The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.


3. ISO 4217, Codes for the representation of currencies and funds.

4. ISO 8601, Data elements and interchange formats — Information interchange — Representation of dates and times.


7. ISO 80000-2, Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology.
3. Terms, definitions, and symbols

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000–2, and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp

Additional terms are defined where they appear in italics or on the left side of a syntax rule. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere.

3.1 access (verb)

⟨execution-time action⟩ to read or modify the value of an object

Note 1 to entry: Where only one of these two actions is meant, “read” or “modify” is used.
Note 2 to entry: “Modify” includes the case where the new value being stored is the same as the previous value.
Note 3 to entry: Expressions that are not evaluated do not access objects.

3.2 alignment

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.3 argument

actual argument
DEPRECATED: actual parameter
expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4 behavior

external appearance or action

3.4.1 implementation-defined behavior

unspecified behavior where each implementation documents how the choice is made

Note 1 to entry: J.3 gives an overview over properties of C programs that lead to implementation-defined behavior.
Example An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2 locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation documents
Note 1 to entry: J.4 gives an overview over properties of C programs that lead to locale-specific behavior.

**EXAMPLE** An example of locale-specific behavior is whether the `islower` function returns true for characters other than the 26 lowercase Latin letters.

### 3.4.3 undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this document imposes no requirements

**Note 1 to entry:** Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

**Note 2 to entry:** J.2 gives an overview over properties of C programs that lead to undefined behavior.

**EXAMPLE** An example of undefined behavior is the behavior on integer overflow.

### 3.4.4 unspecified behavior

behavior, that results from the use of an unspecified value, or other behavior upon which this document provides two or more possibilities and imposes no further requirements on which is chosen in any instance

**Note 1 to entry:** J.1 gives an overview over properties of C programs that lead to unspecified behavior.

**Note 2 to entry:** An example of unspecified behavior is the order in which the arguments to a function are evaluated.

### 3.5 bit

unit of data storage in the execution environment large enough to hold an object that can have one of two values

**Note 1 to entry:** It need not be possible to express the address of each individual bit of an object.

### 3.6 byte

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

**Note 1 to entry:** It is possible to express the address of each individual byte of an object uniquely.

**Note 2 to entry:** A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

### 3.7 character

(a) member of a set of elements used for the organization, control, or representation of data

#### 3.7.1 character

single-byte character (C) bit representation that fits in a byte

#### 3.7.2 multibyte character

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

**Note 1 to entry:** The extended character set is a superset of the basic character set.
3.7.3 wide character
value representable by an object of type `wchar_t`, capable of representing any character in the current locale

3.8 constraint
restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

3.9 correctly rounded result
representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

Note 1 to entry: In this document, when the words “correctly rounded” are not immediately followed by “result”, this is the intended usage.

3.10 diagnostic message
message belonging to an implementation-defined subset of the implementation’s message output

3.11 forward reference
reference to a later subclause of this document that contains additional information relevant to this subclause

3.12 implementation
particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13 implementation limit
restriction imposed upon programs by the implementation

3.14 memory location
either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

Note 1 to entry: Two threads of execution can update and access separate memory locations without interfering with each other.

Note 2 to entry: A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not; or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (nonzero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

Example A structure declared as

```c
struct {
    char a;
    int b:5, c:11,:0, d:8;
    struct { int ee:8; } e;
}
```
contains four separate memory locations: The member a, and bit-fields d and e, ee are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c cannot be concurrently modified, but b and a, for example, can be.

3.15  object  
region of data storage in the execution environment, the contents of which can represent values  

Note 1 to entry: When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16  parameter  
formal parameter  

DEPRECATED: formal argument  
object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.17  recommended practice  
specification that is strongly recommended as being in keeping with the intent of the standard, but that might be impractical for some implementations

3.18  runtime-constraint  
requirement on a program when calling a library function  

Note 1 to entry: Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.  

Note 2 to entry: Implementations that support the extensions in Annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.  

Note 3 to entry: Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.19  value  
precise meaning of the contents of an object when interpreted as having a specific type

3.19.1  implementation-defined value  
unspecified value where each implementation documents how the choice is made

3.19.2  indeterminate value  
either an unspecified value or a trap representation

3.19.3  unspecified value  
valid value of the relevant type where this document imposes no requirements on which value is chosen in any instance  

Note 1 to entry: An unspecified value cannot be a trap representation.
3.19.4 

**trap representation**

an object representation that need not represent a value of the object type

3.19.5 

**perform a trap**

interrupt execution of the program such that no further operations are performed

2 **Note 1 to entry:** In this document, when the word “trap” is not immediately followed by “representation”, this is the intended usage.\(^2\)

3 **Note 2 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

3.20

\[ [x] \]

the least integer greater than or equal to \( x \)

2 **EXAMPLE** \( [2.4] \) is 3, \( [-2.4] \) is -2.

3.21

\[ [x] \]

floor of \( x \)

the greatest integer less than or equal to \( x \)

2 **EXAMPLE** \( [2.4] \) is 2, \( [-2.4] \) is -3.

\(^2\)For example, “Trapping or stopping (if supported) is disabled …” (F.8.2). Note that fetching a trap representation might perform a trap but is not required to (see 6.2.6.1).
4. Conformance

1. In this document, “shall” is to be interpreted as a requirement on an implementation or on a program; conversely, “shall not” is to be interpreted as a prohibition.

2. If a “shall” or “shall not” requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this document by the words “undefined behavior” or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe “behavior that is undefined”.

3. A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

4. The implementation shall not successfully translate a preprocessing translation unit containing a `#error` preprocessing directive unless it is part of a group skipped by conditional inclusion.

5. A strictly conforming program shall use only those features of the language and library specified in this document. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

6. The two forms of conforming implementation are hosted and freestanding. A conforming hosted implementation shall accept any strictly conforming program. A conforming freestanding implementation shall accept any strictly conforming program in which the use of the features specified in the library clause (Clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stdio.h>`, `<stdlib.h>`, and `<stdnoreturn.h>`. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.

7. The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines `__STDC_IEC_60559_BFP__` or `__STDC_IEC_60559_DFP__` may also use features in the contents of the standard headers `<fenv.h>` and `<math.h>` and the numeric conversion functions (7.22.1) of the standard header `<stdlib.h>`. All identifiers that are reserved when `<stdlib.h>` is included in a hosted implementation are reserved when it is included in a freestanding implementation.

8. A conforming program is one that is acceptable to a conforming implementation.

9. An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types `<float.h>` (7.7), alternative spellings `<iso646.h>` (7.9), sizes of integer types `<limits.h>` (7.10), alignment `<stdalign.h>` (7.15), variable arguments `<stdarg.h>` (7.16), boolean type and values `<stdbool.h>` (7.18), common definitions `<stddef.h>` (7.19), integer types `<stdint.h>` (7.20), `<stdnoreturn.h>` (7.23).

---

3) A strictly conforming program can use conditional features (see 6.10.8.3) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```c
#ifdef __STDC_IEC_60559_BFP__ /* FE_UPWARD defined */
  /* ... */
  fesetround(FE_UPWARD);
  /* ... */
#endif
```

4) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this document.

5) Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs can depend upon nonportable features of a conforming implementation.
5. Environment

An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the translation environment and the execution environment in this document. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In this clause, only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

A C program need not all be translated at the same time. The text of the program is kept in units called source files, (or preprocessing files) in this document. A source file together with all the headers and source files included via the preprocessing directive #include is known as a preprocessing translation unit. After preprocessing, a preprocessing translation unit is called a translation unit. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

The precedence among the syntax rules of translation is specified by the following phases.\(^{6}\)

1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

3. The source file is decomposed into preprocessing tokens\(^{7}\) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed, macro invocations are expanded, and _Pragma unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. A #include preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

\(^{6}\)This requires implementations to behave as if these separate phases occur, even though many are typically folded together in practice. Source files, translation units, and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

\(^{7}\)As described in 6.4, the process of dividing a source file’s characters into preprocessing tokens is context-dependent. For example, see the handling of < within a #include preprocessing directive.

\[\]
5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.8

6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

**Forward references:** universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9).

### 5.1.1.3 Diagnostics

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.9

**EXAMPLE** An implementation is required to issue a diagnostic for the translation unit:

```c
char i;
int i;
```

because in those cases where wording in this document describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error is still required to be diagnosed.

### 5.1.2 Execution environments

Two execution environments are defined: freestanding and hosted. In both cases, program startup occurs when a designated C function is called by the execution environment. All objects with static storage duration shall be initialized (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. Program termination returns control to the execution environment.

**Forward references:** storage durations of objects (6.2.4), initialization (6.7.9).

#### 5.1.2.1 Freestanding environment

In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by Clause 4, are implementation-defined.

The effect of program termination in a freestanding environment is implementation-defined.

#### 5.1.2.2 Hosted environment

A hosted environment need not be provided, but shall conform to the following specifications if present.

---

8) An implementation need not convert all non-corresponding source characters to the same execution character.

9) An implementation is encouraged to identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostic messages, often referred to as warnings, as long as a valid program is still correctly translated. It can also successfully translate an invalid program. Annex I lists a few of the more common warnings.
5.1.2.2.1 Program startup

The function called at program startup is named `main`. The implementation declares no prototype for this function. It shall be defined with a return type of `int` and with no parameters:

```
int main(void) { /* ... */ }
```

or with two parameters (referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared):

```
int main(int argc, char *argv[]) { /* ... */ }
```

or equivalent;\(^{10}\) or in some other implementation-defined manner.

If they are declared, the parameters to the `main` function shall obey the following constraints:

---

- The value of `argc` shall be nonnegative.
- `argv[argc]` shall be a null pointer.
- If the value of `argc` is greater than zero, the array members `argv[0]` through `argv[argc-1]` inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.
- If the value of `argc` is greater than zero, the string pointed to by `argv[0]` represents the program name; `argv[0][0]` shall be the null character if the program name is not available from the host environment. If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters.
- The parameters `argc` and `argv` and the strings pointed to by the `argv` array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.
---

5.1.2.2.2 Program execution

In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (Clause 7).

5.1.2.2.3 Program termination

If the return type of the `main` function is a type compatible with `int`, a return from the initial call to the `main` function is equivalent to calling the `exit` function with the value returned by the `main` function as its argument;\(^{11}\) reaching the } that terminates the `main` function returns a value of 0. If the return type is not compatible with `int`, the termination status returned to the host environment is unspecified.

Forward references: definition of terms (7.1.1), the `exit` function (7.22.4.4).

5.1.2.3 Program execution

The semantic descriptions in this document describe the behavior of an abstract machine in which issues of optimization are irrelevant.

Accessing a volatile object, modifying -- An access to an object through the use of an lvalue of volatile-qualified type is a volatile access. A volatile access to an object, modifying a file, or calling a

---

\(^{10}\)Thus, `int` can be replaced by a typedef name defined as `int`, or the type of `argv` can be written as `char ** argv`, and so on.

\(^{11}\)In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in `main` will have ended in the former case, even where they would not have in the latter.
function that does any of those operations are all side effects,\(^\text{12}\) which are changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.

3 **Sequenced before** is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations \(A\) and \(B\), if \(A\) is sequenced before \(B\), then the execution of \(A\) shall precede the execution of \(B\). (Conversely, if \(A\) is sequenced before \(B\), then \(B\) is sequenced after \(A\).) If \(A\) is not sequenced before or after \(B\), then \(A\) and \(B\) are unsequenced. Evaluations \(A\) and \(B\) are indeterminately sequenced when \(A\) is sequenced either before or after \(B\), but it is unspecified which.\(^\text{13}\) The presence of a sequence point between the evaluation of expressions \(A\) and \(B\) implies that every value computation and side effect associated with \(A\) is sequenced before every value computation and side effect associated with \(B\). (A summary of the sequence points is given in Annex C.)

4 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile through volatile access to an object).

5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` are unspecified, as is the state of the dynamic floating-point environment. The value of any object modified by the handler that is neither a lock-free atomic object nor of type `volatile sig_atomic_t` becomes indeterminate when the handler exits, as does the state of the dynamic floating-point environment if it is modified by the handler and not restored to its original state.

6 The least requirements on a conforming implementation are:

- **Accesses to volatile**: Volatile accesses to objects are evaluated strictly according to the rules of the abstract machine.

- At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

- The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

This is the observable behavior of the program.

7 What constitutes an interactive device is implementation-defined.

8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.

9 **EXAMPLE 1** An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword `volatile` would then be redundant.

10 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In

\(^{12}\)The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see Annex F for details. The floating-point environment library `<fenv.h>` provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

\(^{13}\)The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.
this type of implementation, objects referred to by interrupt service routines activated by the signal function would require explicit specification of volatile storage, as well as other implementation-defined restrictions.

11 EXAMPLE 2 In executing the fragment

```c
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the “integer promotions” require that the abstract machine promote the value of each variable to int size and then add the two ints and truncate the sum. Provided the addition of two chars can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

12 EXAMPLE 3 Similarly, in the fragment

```c
float f1, f2;
double d;
/* ... */
f1 = f2 * d;
```

the multiplication can be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if d were replaced by the constant 2.0, which has type double).

13 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit spilling of a register is not permitted to alter the value. Also, an explicit store and load is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```c
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to d1 and d2 are required to have been converted to float.

14 EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.9).

```c
double x, y, z;
/* ... */
x = (x * y) * z; // not equivalent to x *= y * z;
z = (x - y) + y; // not equivalent to z = x;
z = x + x * y; // not equivalent to z = x * (1.0 + y);
y = x / 5.0; // not equivalent to y = x * 0.2;
```

15 EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment

```c
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```c
a = ((a + 32760) + b) + 5;
```

due to the associativity and precedence of these operators. Thus, the result of the sum (a + 32760) is next added to b, and that result is then added to 5 which results in the value assigned to a. On a machine in which overflows produce an explicit trap and in which the range of values representable by an int is [−32768, +32767], the implementation cannot rewrite this expression as

```c
a = ((a + b) + 32765);
```
since if the values for \(a\) and \(b\) were, respectively, \(-32754\) and \(-15\), the sum \(a + b\) would produce a trap while the original expression would not; nor can the expression be rewritten either as

\[
a = (a + 32765) + b;
\]

or

\[
a = (a + (b + 32765));
\]

since the values for \(a\) and \(b\) might have been, respectively, \(4\) and \(-8\) or \(-17\) and \(12\). However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

**EXAMPLE 7** The grouping of an expression does not completely determine its evaluation. In the following fragment

```c
#include <stdio.h>

int sum;
char *p;
/* ... */
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

```c
sum = (((sum * 10) - '0') + (*(p++)) = (getchar()));
```

but the actual increment of \(p\) can occur at any time between the previous sequence point and the next sequence point (the ;), and the call to `getchar` can occur at any point prior to the need of its returned value.

**Forward references:** expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment `<fenv.h>` (7.6), the `signal` function (7.14), files (7.21.3).

### 5.1.2.4 Multi-threaded executions and data races

1. Under a hosted implementation, a program can have more than one **thread of execution** (or **thread**) running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads.

2. The value of an object visible to a thread \(T\) at a particular point is the initial value of the object, a value stored in the object by \(T\), or a value stored in the object by another thread, according to the rules below.

3. **NOTE 1** In some cases, there could instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs.

4. Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location.

5. The library defines a number of **atomic operations** (7.17) and operations on mutexes (7.26.4) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A **synchronization operation** on one or more memory locations is either an **acquire operation**, a **release operation**, both an acquire and release operation, or a **consume operation**. A synchronization operation without an associated memory location is a **fence** and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are **relaxed atomic operations**, which are not synchronization operations, and atomic **read-modify-write operations**, which have special characteristics.

6. **NOTE 2** For example, a call that acquires a mutex will perform an acquire operation on the locations composing the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on \(A\) forces prior side effects on other memory locations to become visible to other threads that later perform an acquire or consume operation on \(A\). Relaxed atomic operations are not included as synchronization operations although, like synchronization operations, they cannot contribute to data races.

---

\(^{14}\) The execution can usually be viewed as an interleaving of all of the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below.
All modifications to a particular atomic object \( M \) occur in some particular total order, called the modification order of \( M \). If \( A \) and \( B \) are modifications of an atomic object \( M \), and \( A \) happens before \( B \), then \( A \) shall precede \( B \) in the modification order of \( M \), which is defined below.

**NOTE 3** This states that the modification orders are expected to respect the “happens before” relation.

**NOTE 4** There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads can observe modifications to different variables in inconsistent orders.

A release sequence headed by a release operation \( A \) on an atomic object \( M \) is a maximal contiguous sub-sequence of side effects in the modification order of \( M \), where the first operation is \( A \) and every subsequent operation either is performed by the same thread that performed the release or is an atomic read-modify-write operation.

Certain library calls synchronize with other library calls performed by another thread. In particular, an atomic operation \( A \) that performs a release operation on an object \( M \) synchronizes with an atomic operation \( B \) that performs an acquire operation on \( M \) and reads a value written by any side effect in the release sequence headed by \( A \).

**NOTE 5** Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation.

**NOTE 6** The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition “reads the value written” by the last mutex release.

An evaluation \( A \) carries a dependency\(^{15} \) to an evaluation \( B \) if:

- the value of \( A \) is used as an operand of \( B \), unless:
  - \( B \) is an invocation of the `kill_dependency` macro,
  - \( A \) is the left operand of a `&&` or `||` operator,
  - \( A \) is the left operand of a `?:` operator, or
  - \( A \) is the left operand of a `,"` operator;

  or

  - \( A \) writes a scalar object or bit-field \( M \), \( B \) reads from \( M \) the value written by \( A \), and \( A \) is sequenced before \( B \), or

  - for some evaluation \( X \), \( A \) carries a dependency to \( X \) and \( X \) carries a dependency to \( B \).

An evaluation \( A \) is dependency-ordered before\(^{16} \) an evaluation \( B \) if:

- \( A \) performs a release operation on an atomic object \( M \), and, in another thread, \( B \) performs a consume operation on \( M \) and reads a value written by any side effect in the release sequence headed by \( A \), or

  - for some evaluation \( X \), \( A \) is dependency-ordered before \( X \) and \( X \) carries a dependency to \( B \).

An evaluation \( A \) inter-thread happens before an evaluation \( B \) if \( A \) synchronizes with \( B \), \( A \) is dependency-ordered before \( B \), or, for some evaluation \( X \):

- \( A \) synchronizes with \( X \) and \( X \) is sequenced before \( B \),

- \( A \) is sequenced before \( X \) and \( X \) inter-thread happens before \( B \), or

- \( A \) inter-thread happens before \( X \) and \( X \) inter-thread happens before \( B \).

\(^{15}\)The “carries a dependency” relation is a subset of the “sequenced before” relation, and is similarly strictly intra-thread.

\(^{16}\)The “dependency-ordered before” relation is analogous to the “synchronizes with” relation, but uses release/consume in place of release/acquire.
NOTE 7 The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced before”, “synchronizes with”, and “dependency-ordered before” relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with “dependency-ordered before” followed by “sequenced before”. The reason for this limitation is that a consume operation participating in a “dependency-ordered before” relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit “inter-thread happens before” to be transitively closed and (2) the “happens before” relation, defined below, provides for relationships consisting entirely of “sequenced before”.

An evaluation $A$ happens before an evaluation $B$ if $A$ is sequenced before $B$ or $A$ inter-thread happens before $B$. The implementation shall ensure that no program execution demonstrates a cycle in the “happens before” relation.

NOTE 8 This cycle would otherwise be possible only through the use of consume operations.

A visible side effect $A$ on an object $M$ with respect to a value computation $B$ of $M$ satisfies the conditions:

- $A$ happens before $B$, and
- there is no other side effect $X$ to $M$ such that $A$ happens before $X$ and $X$ happens before $B$.

The value of a non-atomic scalar object $M$, as determined by evaluation $B$, shall be the value stored by the visible side effect $A$.

NOTE 9 If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.

NOTE 10 This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.

The value of an atomic object $M$, as determined by evaluation $B$, shall be the value stored by some side effect $A$ that modifies $M$, where $B$ does not happen before $A$.

NOTE 11 The set of side effects from which a given evaluation might take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below.

If an operation $A$ that modifies an atomic object $M$ happens before an operation $B$ that modifies $M$, then $A$ shall be earlier than $B$ in the modification order of $M$.

NOTE 12 The requirement above is known as “write-write coherence”.

If a value computation $A$ of an atomic object $M$ happens before a value computation $B$ of $M$, and $A$ takes its value from a side effect $X$ on $M$, then the value computed by $B$ shall either be the value stored by $X$ or the value stored by a side effect $Y$ on $M$, where $Y$ follows $X$ in the modification order of $M$.

NOTE 13 The requirement above is known as “read-read coherence”.

If a value computation $A$ of an atomic object $M$ happens before an operation $B$ on $M$, then $A$ shall take its value from a side effect $X$ on $M$, where $X$ precedes $B$ in the modification order of $M$.

NOTE 14 The requirement above is known as “read-write coherence”.

If a side effect $X$ on an atomic object $M$ happens before a value computation $B$ of $M$, then the evaluation $B$ shall take its value from $X$ or from a side effect $Y$ that follows $X$ in the modification order of $M$.

NOTE 15 The requirement above is known as “write-read coherence”.

NOTE 16 This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are “relaxed” loads. By doing so, it effectively makes the “cache coherence” guarantee provided by most hardware available to C operations.

NOTE 17 The value observed by a load of an atomic object depends on the “happens before” relation, which in turn depends on the values observed by loads of atomic objects. The intended reading is that there exists an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the “happens before” relation derived as described above, satisfy the resulting constraints as imposed here.

The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data
race results in undefined behavior.

NOTE 18 It can be shown that programs that correctly use simple mutexes and `memory_order_seq_cst` operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as "sequential consistency." However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result necessarily has undefined behavior even before such a transformation is applied.

NOTE 19 Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence requirements.

NOTE 20 Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the source character set), and the set interpreted in the execution environment (the execution character set). Each set is further divided into a basic character set, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called extended characters. The combined set is also called the extended character set. The values of the members of the execution character set are implementation-defined.

In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by escape sequences consisting of the backslash \ followed by one or more characters. A byte with all bits set to 0, called the null character, shall exist in the basic execution character set; it is used to terminate a character string.

Both the basic source and basic execution character sets shall have the following members: the 26 uppercase letters of the Latin alphabet

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

the 26 lowercase letters of the Latin alphabet

a b c d e f g h i j k l m
n o p q r s t u v w x y z

the 10 decimal digits

0 1 2 3 4 5 6 7 8 9

the following 29 graphic characters

! " # $ % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~

the space character, and control characters representing horizontal tab, vertical tab, and form feed. The representation of each member of the source and execution basic character sets shall fit in a byte. In both the source and execution basic character sets, the value of each character after 0 in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this document treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the
behavior is undefined.

4 A letter is an uppercase letter or a lowercase letter as defined above; in this document the term does not include other characters that are letters in other alphabets.

5 The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1).

5.2.1.1 Trigraph sequences

Before any other processing takes place, each occurrence of one of the following sequences of three characters (called trigraph sequences)\(^ {17} \) is replaced with the corresponding single character.

\[
??= \quad # \\
??\) \quad } \\
??\( \quad \} \\
??/ \quad \} \\
??< \quad \} \\
\]

No other trigraph sequences exist. Each ? that does not begin one of the trigraphs listed above is not changed.

2 EXAMPLE 1

!??=define arraycheck(a, b) a??(b??) ??!!??! b??(a??)

becomes

#define arraycheck(a, b) a[b] || b[a]

3 EXAMPLE 2

The following source line

\[
printf("Eh??/?n");
\]

becomes (after replacement of the trigraph sequence ??/)

\[
printf("Eh?\n");
\]

5.2.1.2 Multibyte characters

The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:

— The basic character set shall be present and each character shall be encoded as a single byte.

— The presence, meaning, and representation of any additional members is locale-specific.

— A multibyte character set may have a state-dependent encoding, wherein each sequence of multibyte characters begins in an initial shift state and enters other locale-specific shift states when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

— A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.

2 For source files, the following shall hold:

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.

\(^ {17} \) The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set.

§ 5.2.1.2 Environment
— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

1 The active position is that location on a display device where the next character output by the `fputc` function would appear. The intent of writing a printing character (as defined by the `isprint` function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

2 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:

   \a (alert) Produces an audible or visible alert without changing the active position.

   \b (backspace) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.

   \f (form feed) Moves the active position to the initial position at the start of the next logical page.

   \n (new line) Moves the active position to the initial position of the next line.

   \r (carriage return) Moves the active position to the initial position of the current line.

   \t (horizontal tab) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.

   \v (vertical tab) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

3 Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single `char` object. The external representations in a text file need not be identical to the internal representations, and are outside the scope of this document.

Forward references: the `isprint` function (7.4.1.8), the `fputc` function (7.21.7.3).

5.2.3 Signals and interrupts

1 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations’ control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the function image (the instructions that compose the executable representation of a function) on a per-invocation basis.

5.2.4 Environmental limits

1 Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the language-related environmental limits on a conforming implementation; the library-related limits are discussed in Clause 7.

5.2.4.1 Translation limits

1 The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:18)

--- 127 nesting levels of blocks

--- [18] Implementations are encouraged to avoid imposing fixed translation limits whenever possible.
— 63 nesting levels of conditional inclusion
— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or void type in a declaration
— 63 nesting levels of parenthesized declarators within a full declarator
— 63 nesting levels of parenthesized expressions within a full expression
— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)
— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)\(^{19}\)
— 4095 external identifiers in one translation unit
— 511 identifiers with block scope declared in one block
— 4095 macro identifiers simultaneously defined in one preprocessing translation unit
— 127 parameters in one function definition
— 127 arguments in one function call
— 127 parameters in one macro definition
— 127 arguments in one macro invocation
— 4095 characters in a logical source line
— 4095 characters in a string literal (after concatenation)
— 65535 bytes in an object (in a hosted environment only)
— 15 nesting levels for \#included files
— 1023 case labels for a switch statement (excluding those for any nested switch statements)
— 1023 members in a single structure or union
— 1023 enumeration constants in a single enumeration
— 63 levels of nested structure or union definitions in a single struct-declaration-list-member declaration list

5.2.4.2 Numerical limits

An implementation is required to document all the limits specified in this subclause, which are specified in the headers <limits.h> and <float.h>. Additional limits are specified in <stdint.h>. Forward references: integer types <stdint.h> (7.20).

\(^{19}\)See “future language directions” (6.11.3).
5.2.4.2.1 Sizes of integer types `<limits.h>`

The values given below shall be replaced by constant expressions suitable for use in `#if` preprocessing directives.

Moreover, except for `CHAR_BIT` and `MB_LEN_MAX`, and the `width-of-type` macros, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bits for smallest object that is not a bit-field (byte)</td>
<td><code>CHAR_BIT</code> (8)</td>
</tr>
<tr>
<td>Minimum value for an object of type <code>signed char</code></td>
<td><code>SCHAR_MIN</code> (-127) (\text{\textendash} (2^7 - 1))</td>
</tr>
<tr>
<td>Maximum value for an object of type <code>signed char</code></td>
<td><code>SCHAR_MAX</code> (+127) (2^7 - 1)</td>
</tr>
<tr>
<td>Width of type <code>signed char</code></td>
<td><code>SCHAR_WIDTH</code> (8)</td>
</tr>
<tr>
<td>Maximum value for an object of type <code>unsigned char</code></td>
<td><code> UCHAR_MAX</code> (255) (2^8 - 1)</td>
</tr>
<tr>
<td>Width of type <code>unsigned char</code></td>
<td><code> UCHAR_WIDTH</code> (8)</td>
</tr>
<tr>
<td>Minimum value for an object of type <code>char</code></td>
<td><code>CHAR_MIN</code> see below</td>
</tr>
<tr>
<td>Maximum value for an object of type <code>char</code></td>
<td><code>CHAR_MAX</code> see below</td>
</tr>
<tr>
<td>Width of type <code>char</code></td>
<td><code>CHAR_WIDTH</code> (8)</td>
</tr>
<tr>
<td>Maximum number of bytes in a multibyte character, for any supported locale</td>
<td><code>MB_LEN_MAX</code> (1)</td>
</tr>
<tr>
<td>Minimum value for an object of type <code>short int</code></td>
<td><code>SHRT_MIN</code> (-32767) (\text{\textendash} (2^{15} - 1))</td>
</tr>
<tr>
<td>Maximum value for an object of type <code>short int</code></td>
<td>see below</td>
</tr>
<tr>
<td>Identifier</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>Maximum value for an object of type <code>short int</code></td>
</tr>
<tr>
<td>SHRT_WIDTH</td>
<td>Width of type <code>short int</code></td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>Maximum value for an object of type <code>unsigned short int</code></td>
</tr>
<tr>
<td>USHRT_WIDTH</td>
<td>Width of type <code>unsigned short int</code></td>
</tr>
<tr>
<td>INT_MIN</td>
<td>Minimum value for an object of type <code>int</code></td>
</tr>
<tr>
<td>INT_MAX</td>
<td>Maximum value for an object of type <code>int</code></td>
</tr>
<tr>
<td>INT_WIDTH</td>
<td>Width of type <code>int</code></td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>Maximum value for an object of type <code>unsigned int</code></td>
</tr>
<tr>
<td>UINT_WIDTH</td>
<td>Width of type <code>unsigned int</code></td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>Minimum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>Maximum value for an object of type <code>long int</code></td>
</tr>
<tr>
<td>LONG_WIDTH</td>
<td>Width of type <code>long int</code></td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>Maximum value for an object of type <code>unsigned long int</code></td>
</tr>
<tr>
<td>ULONG_WIDTH</td>
<td>Width of type <code>unsigned long int</code></td>
</tr>
</tbody>
</table>
— minimum value for an object of type \texttt{long long int}
\begin{verbatim}
LLONG_MIN -922372836854775807 // \(-2^{63} - 1\)
\end{verbatim}

— maximum value for an object of type \texttt{long long int}
\begin{verbatim}
LLONG_MAX +922372836854775807 // \(2^{63} - 1\)
\end{verbatim}

— width of type \texttt{long long int}
\begin{verbatim}
LLONG_WIDTH 64
\end{verbatim}

— maximum value for an object of type \texttt{unsigned long long int}
\begin{verbatim}
ULLONG_MAX 18446744073709551615 // \(2^{64} - 1\)
\end{verbatim}

— width of type \texttt{unsigned long long int}
\begin{verbatim}
ULLONG_WIDTH 64
\end{verbatim}

2. If an object of type \texttt{char} can hold negative values, the value of \texttt{CHAR_MIN} shall be the same as that of \texttt{SCHAR_MIN} and the value of \texttt{CHAR_MAX} shall be the same as that of \texttt{SCHAR_MAX}. Otherwise, the value of \texttt{CHAR_MIN} shall be 0 and the value of \texttt{CHAR_MAX} shall be the same as that of \texttt{UCHAR_MAX}.\(^{20}\) The value \texttt{UCHAR_MAX} shall equal \(2^{\text{CHAR_BIT}} - 1\).

Forward references: representations of types (6.2.6), conditional inclusion (6.10.1).

5.2.4.2.2 Characteristics of floating types \texttt{<float.h>}

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation’s floating-point arithmetic.\(^{21}\) An implementation that defines \texttt{__STDC_IEC_60559_BFP__} or \texttt{__STDC_IEC_559__} shall implement floating point types and arithmetic conforming to IEC 60559 as specified in Annex F. An implementation that defines \texttt{__STDC_IEC_60559_COMPLEX__} or \texttt{__STDC_IEC_559_COMPLEX__} shall implement complex types and arithmetic conforming to IEC 60559 as specified in Annex G.

2. The following parameters are used to define the model for each floating-point type:
\begin{itemize}
\item \texttt{s} sign (\(\pm 1\))
\item \texttt{b} base or radix of exponent representation (an integer \(> 1\))
\item \texttt{e} exponent (an integer between a minimum \(e_{\text{min}}\) and a maximum \(e_{\text{max}}\))
\item \texttt{p} precision (the number of base-\(b\) digits in the significand)
\item \texttt{f_k} nonnegative integers less than \(b\) (the significand digits)
\end{itemize}

For each floating-point type, the parameters \(b, p, e_{\text{min}}, e_{\text{max}}\) are fixed constants.

3. For each floating-point type, a floating-point number \((x)\) is defined by the following model:
\[ x = s b^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\text{min}} \leq e \leq e_{\text{max}} \]

4. In addition to normalized floating-point numbers, floating types shall be able to represent zero (all \(f_k \geq 0\)) and all normalized floating-point numbers \((f_1 > 0 \text{ if } x \neq 0)\) and all possible \(k\) digits and \(e\) exponents result in values representable in the type. In addition, floating types may be able to

\(^{20}\)See 6.2.5.

\(^{21}\)The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical.
contain other kinds of floating-point numbers, such as \(^{22)}\) such as negative zero, subnormal floating-point numbers \((x \neq 0, e = e_{\text{min}}, f_1 = 0)\) and unnormalized floating-point numbers \((x \neq 0, e > e_{\text{min}}, f_1 = 0)\), and values that are not floating-point numbers, such as infinities and NaNs. A NaN is an encoding of a value signifying Not-a-Number. A quiet NaN propagates through almost every arithmetic operation without raising a floating-point exception; a signaling NaN generally raises a floating-point exception when occurring as an arithmetic operand.\(^{23)}\)

An implementation may give zero and values that are not floating-point numbers (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in this document to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

An implementation may prefer particular representations of values that have multiple representations in a floating type, 6.2.6.1 not withstanding.\(^{24)}\) The preferred representations of a floating type, including unique representations of values in the type, are called canonical. A floating type may also contain non-canonical representations, for example, redundant representations of some or all of its values, or representations that are extraneous to the floating-point model.\(^{25)}\) Typically, floating-point operations deliver results with canonical representations. IEC 60559 operations deliver results with canonical representations, unless specified otherwise.

The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. In addition, if negative infinity is representable in a type, the range of that type is extended to all negative real numbers; likewise, if positive infinity is representable in a type, the range of that type is extended to all positive real numbers.

The accuracy of the floating-point operations \((+, -, \times, /)\) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown. Decimal floating-point operations have stricter requirements.

All integer values in the `<float.h>` header, except `FLT_ROUNDS`, shall be constant expressions suitable for use in `#if` preprocessing directives; all floating values shall be constant expressions. All except `DECIMAL_DIG` and `DECIMAL_DIG_F` (`F5`), `DECIMAL_DIG`, `DEC_EVAL_METHOD`, `FLT_EVAL_METHOD`, `FLT_RADIX`, and `FLT_ROUNDS` have separate names for all three floating-point types. The floating-point model representation is provided for all values except `DEC_EVAL_METHOD`, `FLT_EVAL_METHOD` and `FLT_ROUNDS`.

The remainder of this subclause specifies characteristics of standard floating types.

The rounding mode for floating-point addition for standard floating types is characterized by the implementation-defined value of `FLT_ROUNDS`. Evaluation of `FLT_ROUNDS` correctly reflects any execution-time change of rounding mode through the function `fesetround` in `<fenv.h>`.

- **1** indeterminable
- **0** toward zero
- **1** to nearest, ties to even
- **2** toward positive infinity
- **3** toward negative infinity

\(^{22)}\) Some implementations have types that include finite numbers with extra range and/or precision that are not covered by the model.

\(^{23)}\) IEC 60559:1989 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to values with similar behavior.

\(^{24)}\) The library operations `iscanonical` and `canonicalize` distinguish canonical (preferred) representations, but this distinction alone does not imply that canonical and non-canonical representations are of different values.

\(^{25)}\) Some of the values in the IEC 60559 decimal formats have non-canonical representations (as well as a canonical representation).
All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

Except for assignment and cast (which remove all extra range and precision), the values of floating type yielded by operators with floating operands and values subject to the usual arithmetic conversions and, including the values yielded by the implicit conversion of operands, and the values of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats such a format is called an evaluation format. In all cases, assignment and cast operators yield values in the format of the type. The extent to which evaluation formats are used is characterized by the implementation-defined value of `FLT_EVAL_METHOD`:

-1 indeterminable;
0 evaluate all operations and constants just to the range and precision of the type;
1 evaluate operations and constants of type `float` and `double` to the range and precision of the `double` type, evaluate `long double` operations and constants to the range and precision of the `long double` type;
2 evaluate all operations and constants to the range and precision of the `long double` type.

All other negative values for `FLT_EVAL_METHOD` characterize implementation-defined behavior. The value of `FLT_EVAL_METHOD` does not characterize values returned by function calls (see 6.8.6.4, F.6).

The presence or absence of subnormal numbers is characterized by the implementation-defined values of `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM`, and `LDBL_HAS_SUBNORM`:

-1 indeterminable
0 absent (type does not support subnormal numbers)
1 present (type does support subnormal numbers)

The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

- radix of exponent representation, $b$

<table>
<thead>
<tr>
<th><code>FLT_RADIX</code></th>
<th>2</th>
</tr>
</thead>
</table>

- number of base-$b$ digits in the floating-point significand, $p$

<table>
<thead>
<tr>
<th><code>FLT_MANT_DIG</code></th>
<th><code>DBL_MANT_DIG</code></th>
<th><code>LDBL_MANT_DIG</code></th>
</tr>
</thead>
</table>

---

26) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if `FLT_EVAL_METHOD` is 1, then the product of two `float _Complex` operands is represented in the `double _Complex` format, and its parts are evaluated to `double`.

27) Characterization as indeterminable is intended if floating-point operations do not consistently interpret subnormal representations as zero, nor as nonzero.

28) Characterization as absent is intended if no floating-point operations produce subnormal results from non-subnormal inputs, even if the type format includes representations of subnormal numbers.
— number of decimal digits, \( n \), such that any floating-point number with \( p \) radix \( b \) digits can be rounded to a floating-point number with \( n \) decimal digits and back again without change to the value,

\[
\begin{cases} 
  p \log_{10} b & \text{if } b \text{ is a power of } 10 \\
  \lceil 1 + p \log_{10} b \rceil & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>DECIMAL_DIG</th>
<th>10</th>
</tr>
</thead>
</table>

This is an obsolescent feature, see 7.31.8.

— number of decimal digits, \( n \), such that any floating-point number in the widest supported floating type with \( p_{\text{max}} \) radix \( b \) digits can be rounded to a floating-point number with \( n \) decimal digits and back again without change to the value,

\[
\begin{cases} 
  p_{\text{max}} \log_{10} b & \text{if } b \text{ is a power of } 10 \\
  \lceil 1 + p_{\text{max}} \log_{10} b \rceil & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>DECIMAL_DIG</th>
<th>10</th>
</tr>
</thead>
</table>

— number of decimal digits, \( q \), such that any floating-point number with \( q \) decimal digits can be rounded into a floating-point number with \( p \) radix \( b \) digits and back again without change to the \( q \) decimal digits,

\[
\begin{cases} 
  p \log_{10} b & \text{if } b \text{ is a power of } 10 \\
  \lceil (p - 1) \log_{10} b \rceil & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>FLT_DIG</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_DIG</td>
<td>10</td>
</tr>
<tr>
<td>LDBL_DIG</td>
<td>10</td>
</tr>
</tbody>
</table>

— minimum negative integer such that \( \text{FLT\_RADIX} \) raised to one less than that power is a normalized floating-point number, \( e_{\text{min}} \)

<table>
<thead>
<tr>
<th>FLT_MIN_EXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_MIN_EXP</td>
</tr>
<tr>
<td>LDBL_MIN_EXP</td>
</tr>
</tbody>
</table>

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, \( \lceil \log_{10} b^{e_{\text{min}} - 1} \rceil \)

| FLT_MIN_10_EXP | -37 |
| DBL_MIN_10_EXP | -37 |
| LDBL_MIN_10_EXP | -37 |

— maximum integer such that \( \text{FLT\_RADIX} \) raised to one less than that power is a representable finite floating-point number, \( e_{\text{max}} \)

<table>
<thead>
<tr>
<th>FLT_MAX_EXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_MAX_EXP</td>
</tr>
<tr>
<td>LDBL_MAX_EXP</td>
</tr>
</tbody>
</table>
— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, \(\lfloor \log_{10}(1 - b^{-p})b^{e_{\text{max}}} \rfloor \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_MAX_10_EXP</td>
<td>+37</td>
</tr>
<tr>
<td>DBL_MAX_10_EXP</td>
<td>+37</td>
</tr>
<tr>
<td>LDBL_MAX_10_EXP</td>
<td>+37</td>
</tr>
</tbody>
</table>

15 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown:

— maximum representable finite floating-point number, \(r; \text{if that number is normalized, its value is } (1 - b^{-p})b^{e_{\text{max}}} \)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_MAX</td>
<td>1E+37</td>
</tr>
<tr>
<td>DBL_MAX</td>
<td>1E+37</td>
</tr>
<tr>
<td>LDBL_MAX</td>
<td>1E+37</td>
</tr>
</tbody>
</table>

— maximum normalized floating-point number, \((1 - b^{-p})b^{e_{\text{max}}})\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_NORM_MAX</td>
<td>1E+37</td>
</tr>
<tr>
<td>DBL_NORM_MAX</td>
<td>1E+37</td>
</tr>
<tr>
<td>LDBL_NORM_MAX</td>
<td>1E+37</td>
</tr>
</tbody>
</table>

16 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

— the difference between 1 and the least normalized value greater than 1 that is representable in the given floating-point type, \(b^{1-p}\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_EPSILON</td>
<td>1E-5</td>
</tr>
<tr>
<td>DBL_EPSILON</td>
<td>1E-9</td>
</tr>
<tr>
<td>LDBL_EPSILON</td>
<td>1E-9</td>
</tr>
</tbody>
</table>

— minimum normalized positive floating-point number, \(b^{e_{\text{min}}-1}\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_MIN</td>
<td>1E-37</td>
</tr>
<tr>
<td>DBL_MIN</td>
<td>1E-37</td>
</tr>
<tr>
<td>LDBL_MIN</td>
<td>1E-37</td>
</tr>
</tbody>
</table>

— minimum positive floating-point number\(^{29}\)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_TRUE_MIN</td>
<td>1E-37</td>
</tr>
<tr>
<td>DBL_TRUE_MIN</td>
<td>1E-37</td>
</tr>
<tr>
<td>LDBL_TRUE_MIN</td>
<td>1E-37</td>
</tr>
</tbody>
</table>

**Recommended practice**

17 Conversion from (at least) double to decimal with DECIMAL_DIG between real floating type and decimal character sequence with at most \(T\)_DECIMAL_DIG digits should be correctly rounded, where \(T\) is the macro prefix for the type. This assures conversion from real floating type to decimal character sequence with \(T\)_DECIMAL_DIG digits and back should be using to-nearest rounding, is the identity function.

18 EXAMPLE 1 The following describes an artificial floating-point representation that meets the minimum requirements of this document, and the appropriate values in a <float.h> header for type float:

\(^{29}\)If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.
x = s16^6 \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \leq e \leq +32

\begin{array}{l}
\text{FLT_RADIX} & 16 \\
\text{FLT_MANT_DIG} & 6 \\
\text{FLT_EPSILON} & 9.53674316E-07F \\
\text{FLT_DECIMAL_DIG} & 9 \\
\text{FLT_DIG} & 6 \\
\text{FLT_MIN_EXP} & -31 \\
\text{FLT_MIN} & 2.93873588E-39F \\
\text{FLT_MIN_10_EXP} & -38 \\
\text{FLT_MAX} & 3.40282347E+38F \\
\text{FLT_MAX_10_EXP} & +38 \\
\end{array}

\textbf{EXAMPLE 2} The following describes floating-point representations that also meet the requirements for single-precision and double-precision numbers in IEC 60559,\textsuperscript{30} and the appropriate values in a <float.h> header for types \texttt{float} and \texttt{double}:

\[x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \leq e \leq +128\]
\[x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \leq e \leq +1024\]

If a type wider than \texttt{double} were supported, then \texttt{DECIMAL_DIG} would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then \texttt{DECIMAL_DIG} would be 21.

\begin{array}{l}
\text{FLT_RADIX} & 2 \\
\text{FLT_MANT_DIG} & 24 \\
\text{FLT_EPSILON} & 1.19209290E-07F // decimal constant \\
\text{FLT_EPSILON} & 0X1P-23F // hex constant \\
\text{FLT_DECIMAL_DIG} & 9 \\
\text{FLT_DIG} & 6 \\
\text{FLT_MIN_EXP} & -125 \\
\text{FLT_MIN} & 1.17549435E-38F // decimal constant \\
\text{FLT_MIN} & 0X1P-126F // hex constant \\
\text{FLT_TRUE_MIN} & 1.40129846E-45F // decimal constant \\
\text{FLT_TRUE_MIN} & 0X1P-149F // hex constant \\
\text{FLT_HAS_SUBNORM} & 1 \\
\text{FLT_MIN_10_EXP} & -37 \\
\text{FLT_MAX} & 3.40282347E+38F // decimal constant \\
\text{FLT_MAX} & 0X1.fffffp0127F // hex constant \\
\text{FLT_MAX_10_EXP} & +37 \\
\text{DBL_MANT_DIG} & 53 \\
\text{DBL_EPSILON} & 2.2204460492503131E-16 // decimal constant \\
\text{DBL_EPSILON} & 0X1P-52 // hex constant \\
\text{DBL_DECIMAL_DIG} & 17 \\
\text{DBL_DIG} & 15 \\
\text{DBL_MIN_EXP} & -1021 \\
\text{DBL_MIN} & 2.2250738585072014E-308 // decimal constant \\
\text{DBL_MIN} & 0X1P-1022 // hex constant \\
\text{DBL_TRUE_MIN} & 4.9406564584124654E-324 // decimal constant \\
\text{DBL_TRUE_MIN} & 0X1P-1074 // hex constant \\
\text{DBL_HAS_SUBNORM} & 1 \\
\text{DBL_MIN_10_EXP} & -307 \\
\text{DBL_MAX} & +1024 \\
\text{DBL_MAX} & 1.7976931348623157E+308 // decimal constant \\
\text{DBL_MAX} & 0X1.fffffffffffffffP1023 // hex constant \\
\end{array}

\textsuperscript{30}The floating-point model in that standard sums powers of \texttt{b} from zero, so the values of the exponent limits are one less than shown here.
Forward references: conditional inclusion (6.10.1), predefined macro names (6.10.8), complex arithmetic <complex.h> (7.3), extended multibyte and wide character utilities <wchar.h> (7.29), floating-point environment <fenv.h> (7.6), general utilities <stdlib.h> (7.22), input/output <stdio.h> (7.21), mathematics <math.h> (7.12), IEC 60559 floating-point arithmetic (Annex F), IEC 60559-compatible complex arithmetic (Annex G).}

5.2.4.2.3 Characteristics of decimal floating types in <float.h>

This subclause specifies macros in <float.h> that provide characteristics of decimal floating types in terms of the model presented in 5.2.4.2.2. An implementation that does not support decimal floating types shall not provide these macros. The prefixes DEC32_, DEC64_, and DEC128_ denote the types _Decimal32, _Decimal64, and _Decimal128 respectively.

1 DEC_EVAL_METHOD is the decimal floating-point analog of FLT_EVAL_METHOD (5.2.4.2.2). Its implementation-defined value characterizes the use of evaluation formats for decimal floating types:

-1 indeterminable;
0 evaluate all operations and constants just to the range and precision of the type;
1 evaluate operations and constants of type _Decimal32 and _Decimal64 to the range and precision of the _Decimal64 type, evaluate _Decimal128 operations and constants to the range and precision of the _Decimal128 type;
2 evaluate all operations and constants to the range and precision of the _Decimal128 type.

3 The integer values given in the following lists shall be replaced by constant expressions suitable for use in #if preprocessing directives:

- radix of exponent representation, b (=10)

  For the standard floating types, this value is implementation-defined and is specified by the macro FLT_RADIX. For the decimal floating types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever FLT_RADIX appears in a description of a function that has versions that operate on decimal floating types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than FLT_RADIX.

- number of digits in the coefficient

| DEC32_MANT_DIG | 7 |
| DEC64_MANT_DIG | 16 |
| DEC128_MANT_DIG | 34 |

- minimum exponent

| DEC32_MIN_EXP | -94 |
| DEC64_MIN_EXP | -382 |
| DEC128_MIN_EXP | -6142 |

- maximum exponent

| DEC32_MAX_EXP | 97 |
| DEC64_MAX_EXP | 385 |
| DEC128_MAX_EXP | 6145 |

- maximum representable finite decimal floating-point number (there are 6, 15 and 33 9’s after the decimal points respectively)
the difference between 1 and the least value greater than 1 that is representable in the given floating type

- minimum normalized positive decimal floating-point number

- minimum positive subnormal decimal floating-point number

For decimal floating-point arithmetic, it is often convenient to consider an alternate equivalent model where the significand is represented with integer rather than fraction digits. With \( s, b, c, p, \) and \( f_k \) as defined in 5.2.4.2.2, a floating-point number \( x \) is defined by the model:

\[
x = s \cdot b^{(e-p)} \sum_{k=1}^{p} f_k \cdot b^{(p-k)}
\]

With \( b \) fixed to 10, a decimal floating-point number \( x \) is thus:

\[
x = s \cdot 10^{(e-p)} \sum_{k=1}^{p} f_k \cdot 10^{(p-k)}
\]

The quantum exponent is \( q \equiv e - p \) and the coefficient is \( c \equiv f_1 f_2 \cdots f_p \), which is an integer between 0 and \( 10^{(p-1)} \) inclusive. Thus, \( x = s \cdot c \cdot 10^p \) is represented by the triple of integers \( (s, c, q) \). The quantum of \( x \) is \( 10^q \), which is the value of a unit in the last place of the coefficient.

Quantum exponent ranges

<table>
<thead>
<tr>
<th>Type</th>
<th>Decimal32</th>
<th>Decimal64</th>
<th>Decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Quantum Exponent ((q_{\text{max}}))</td>
<td>90</td>
<td>369</td>
<td>6111</td>
</tr>
<tr>
<td>Minimum Quantum Exponent ((q_{\text{min}}))</td>
<td>-101</td>
<td>-398</td>
<td>-6176</td>
</tr>
</tbody>
</table>

For binary floating-point arithmetic following IEC 60559, representations in the model described in 5.2.4.2.2 that have the same numerical value are indistinguishable in the arithmetic. However, for decimal floating-point arithmetic, representations that have the same numerical value but different quantum exponents, e.g., \( (1, 10, -1) \) representing \( 1.0 \) and \( (1, 100, -2) \) representing \( 1.00 \), are distinguishable. To facilitate exact fixed-point calculation, operation results that are of decimal floating type have a preferred quantum exponent, as specified in IEC 60559, which is determined by the quantum exponents of the operands if they have decimal floating types (or by specific

Environment § 5.2.4.2.3
rules for conversions from other types). The table below gives rules for determining preferred quantum exponents for results of IEC 60559 operations, and for other operations specified in this document. When exact, these operations produce a result with their preferred quantum exponent, or as close to it as possible within the limitations of the type. When inexact, these operations produce a result with the least possible quantum exponent. For example, the preferred quantum exponent for addition is the minimum of the quantum exponents of the operands. Hence 
\((1,123, -2) + (1,4000, -3) = (1,5230, -3)\) or 
\((1,23 + 4,000) = 5,230.\)

The following table shows, for each operation delivering a result in decimal floating-point format, how the preferred quantum exponents of the operands, \(Q(x), Q(y), \text{ etc.}\), determine the preferred quantum exponent of the operation result.

### Preferred quantum exponents

<table>
<thead>
<tr>
<th>Operation</th>
<th>Preferred quantum exponent of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>roundeven, round, trunc, ceil, floor, rint, nearbyint</td>
<td>(\max(Q(x), 0))</td>
</tr>
<tr>
<td>nextup, nextdown, nextafter, nexttoward</td>
<td>least possible</td>
</tr>
<tr>
<td>remainder</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>fmin, fmax, fminmag, fmaxmag</td>
<td>(Q(x)) if (x) gives the result, (Q(y)) if (y) gives the result</td>
</tr>
<tr>
<td>scalbn, scalbln</td>
<td>(Q(x) + n)</td>
</tr>
<tr>
<td>ldexp</td>
<td>(Q(x) + p)</td>
</tr>
<tr>
<td>logb</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+, d32add, d64add</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>-, d32sub, d64sub</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>*, d32mul, d64mul</td>
<td>(Q(x) + Q(y))</td>
</tr>
<tr>
<td>/, d32div, d64div</td>
<td>(Q(x) - Q(y))</td>
</tr>
<tr>
<td>sqrt, d32sqrt, d64sqrt</td>
<td>(\lfloor Q(x)/2\rfloor)</td>
</tr>
<tr>
<td>fma, d32fma, d64fma</td>
<td>(\min(Q(x) + Q(y), Q(z)))</td>
</tr>
<tr>
<td>conversion from integer type</td>
<td>0</td>
</tr>
<tr>
<td>exact conversion from non-decimal floating type</td>
<td>0</td>
</tr>
<tr>
<td>inexact conversion from non-decimal floating type</td>
<td>least possible</td>
</tr>
<tr>
<td>conversion between decimal floating types</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>(*_x) returned by canonicalize</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>strto, wcsto, scanf, floating constants of decimal floating type</td>
<td>see 7.22.1.6</td>
</tr>
<tr>
<td>(-_)</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>fabs</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>copysign</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>quantize</td>
<td>(Q(y))</td>
</tr>
<tr>
<td>quantum</td>
<td>(Q(x))</td>
</tr>
<tr>
<td>*encptr returned by encodedec, encodebin</td>
<td>(Q(*xptr))</td>
</tr>
<tr>
<td>*xptr returned by decodedec, decodebin</td>
<td>(Q(*xptr))</td>
</tr>
<tr>
<td>fmod</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>fdim</td>
<td>(\min((Q(x), Q(y))) if (x &gt; y), 0) if (x \leq y)</td>
</tr>
<tr>
<td>cbrt</td>
<td>(\lfloor Q(x)/3\rfloor)</td>
</tr>
<tr>
<td>hypot</td>
<td>(\min(Q(x), Q(y)))</td>
</tr>
<tr>
<td>pow</td>
<td>(y \times Q(x))</td>
</tr>
<tr>
<td>mod</td>
<td>(Q(value))</td>
</tr>
<tr>
<td>*iptr returned by mod</td>
<td>(\max(Q(value), 0))</td>
</tr>
</tbody>
</table>
frexp

\(Q(\text{value})\) if \(\text{value} = 0\), 
\(-(\text{length of coefficient of value})\) otherwise

<table>
<thead>
<tr>
<th>*res__ returned__ by__ setpayload__</th>
<th>setpayloadsig</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 if (\text{pl}) does not represent a valid payload, not applicable otherwise (NaN returned).</td>
<td></td>
</tr>
</tbody>
</table>

getpayload

0 if \(\ast x\) is a NaN, unspecified otherwise.

compoundn

\([n \times \text{min}(0, Q(x))]\)

pown

\([n \times Q(x)]\)

powr

\([y \times Q(x)]\)

rootn

\([Q(x)/n]\)

rsqrt

\(-[Q(x)/2]\)

transcendental functions

0

A function family listed in the table above indicates the functions for all decimal floating types, where the function family is represented by the name of the functions without a suffix. For example, ceil indicates the functions ceild32, ceild64, and ceild128.

Forward references: extended multibyte and wide character utilities <wchar.h> (7.29), floating-point environment <fenv.h> (7.6), general utilities <stdlib.h> (7.22), input/output <stdio.h> (7.21), mathematics <math.h> (7.12), type-generic mathematics <tgmath.h> (7.25), IEC 60559 floating-point arithmetic (Annex F).
6. Language

6.1 Notation

In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by italic type, and literal words and character set members (terminals) by bold type. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that

\[
\{expression opt\} \{ expression opt \}
\]

indicates an optional expression enclosed in braces.

When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an enumeration constant. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its scope. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A function prototype is a declaration of a function that declares the types of its parameters.)

A label name is the only kind of identifier that has function scope. It can be used (in a goto statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement).

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has file scope, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has function prototype scope, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the inner scope) will end strictly before the scope of the other entity (the outer scope). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden (and not visible) within the inner scope.

Unless explicitly stated otherwise, where this document uses the term “identifier” to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

Two identifiers have the same scope if and only if their scopes terminate at the same point.

Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that
begins just after the completion of its declarator.

As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called linkage. There are three kinds of linkage: external, internal, and none.

In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with external linkage denotes the same object or function. Within one translation unit, each declaration of an identifier with internal linkage denotes the same object or function. Each declaration of an identifier with no linkage denotes a unique entity.

If the declaration of a file scope identifier for an object or a function contains the storage-class specifier static, the identifier has internal linkage.

For an identifier declared with the storage-class specifier extern in a scope in which a prior declaration of that identifier is visible, if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier extern. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier extern.

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:

— label names (disambiguated by the syntax of the label declaration and use);

— the tags of structures, unions, and enumerations (disambiguated by following any of the keywords struct, union, or enum);

— the members of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . or -> operator);

— all other identifiers, called ordinary identifiers (declared in ordinary declarators or as enumeration constants).

31) There is no linkage between different identifiers.
32) A function declaration can contain the storage-class specifier static only if it is at file scope; see 6.7.1.
33) As specified in 6.2.1, the later declaration might hide the prior declaration.
34) There is only one name space for tags even though three are possible.
Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the goto statement (6.8.6.1).

6.2.4 Storage durations of objects

1 An object has a storage duration that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,\(^{35}\) and retains its last-stored value throughout its lifetime.\(^{36}\) If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

3 An object whose identifier is declared without the storage-class specifier _Thread_local, and either with external or internal linkage or with the storage-class specifier static, has static storage duration. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier _Thread_local has thread storage duration. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 An object whose identifier is declared with no linkage and without the storage-class specifier static has automatic storage duration, as do some compound literals. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.\(^{37}\) If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and temporary lifetime.\(^{38}\) Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime results in undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. Such an object need not have a unique address.

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), initialization (6.7.9), statements (6.8), effective type (6.5).

---

\(^{35}\) The term “constant address” means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

\(^{36}\) In the case of a volatile object, the last store need not be explicit in the program.

\(^{37}\) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

\(^{38}\) The address of such an object is taken implicitly when an array member is accessed.
6.2.5 Types

1 The meaning of a value stored in an object or returned by a function is determined by the type of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into object types (types that describe objects) and function types (types that describe functions). At various points within a translation unit an object type may be incomplete (lacking sufficient information to determine the size of objects of that type) or complete (having sufficient information).

2 An object declared as type _Bool is large enough to store the values 0 and 1.

3 An object declared as type char is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a char object, its value is guaranteed to be nonnegative. If any other character is stored in a char object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

4 There are five standard signed integer types, designated as signed char, short int, int, long int, and long long int. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined extended signed integer types. The standard and extended signed integer types are collectively called signed integer types.

5 An object declared as type signed char occupies the same amount of storage as a “plain” char object. A “plain” int object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range INT_MIN to INT_MAX as defined in the header <limits.h>).

6 For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword unsigned) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type _Bool and the unsigned integer types that correspond to the standard signed integer types are the standard unsigned integer types. The unsigned integer types that correspond to the extended signed integer types are the extended unsigned integer types. The standard and extended unsigned integer types are collectively called unsigned integer types.

7 The standard signed integer types and standard unsigned integer types are collectively called the standard integer types; the extended signed integer types and extended unsigned integer types are collectively called the extended integer types.

8 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same. A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

10 There are three standard floating types, designated as float, double, and long double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double.

---

39) A type can be incomplete or complete throughout an entire translation unit, or it can change states at different points within a translation unit.

40) Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

41) Therefore, any statement in this document about signed integer types also applies to the extended signed integer types.

42) Therefore, any statement in this document about unsigned integer types also applies to the extended unsigned integer types.

43) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

44) See “future language directions” (6.11.1).
There are three decimal floating types, designated as _Decimal32, _Decimal64, and _Decimal128. Respectively, they have the IEC 60559 formats: decimal32, \texttt{decimal64}, and \texttt{decimal128}. Decimal floating types are real floating types.

The standard floating types and the decimal floating types are collectively called the real floating types.

There are three complex types, designated as \texttt{float _Complex}, \texttt{double _Complex}, and \texttt{long double _Complex}. \texttt{Complex} types are a conditional feature that implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively called the floating types.

For each floating type there is a corresponding real type, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword _Complex from the type name.

Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

The type char, the signed and unsigned integer types, and the floating types are collectively called the basic types. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

The three types char, signed char, and unsigned char are collectively called the character types. The implementation shall define char to have the same range, representation, and behavior as either signed char or unsigned char.

An enumeration comprises a set of named integer constant values. Each distinct enumeration constitutes a different enumerated type.

The type char, the signed and unsigned integer types, and the enumerated types are collectively called integer types. The integer and real floating types are collectively called real types.

Integer and floating types are collectively called arithmetic types. Each arithmetic type belongs to one type domain: the real type domain comprises the real types, the complex type domain comprises the complex types.

The void type comprises an empty set of values; it is an incomplete object type that cannot be completed.

Any number of derived types can be constructed from the object and function types, as follows:

- An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the element type. The element type shall be complete whenever the array type is specified. Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is \( T \), the array type is sometimes called “array of \( T \)”. The construction of an array type from an element type is called “array type derivation”.

- A structure type describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

- A union type describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

---

\(^{45}\) IEC 60559 specifies decimal32 as a data-interchange format that does not require arithmetic support; however, _Decimal32 is a fully supported arithmetic type.

\(^{46}\) A specification for imaginary types is in Annex G.

\(^{47}\) An implementation can define new keywords that provide alternative ways to designate a basic (or any other) type; this does not violate the requirement that all basic types be different. Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

\(^{48}\) CHAR_MIN, defined in \texttt{<limits.h>}, will have one of the values 0 or SCHAR_MIN, and this can be used to distinguish the two options. Irrespective of the choice made, char is a separate type from the other two and is not compatible with either.

\(^{49}\) Annex H documents the extent to which the C language supports the ISO/IEC 10967-1 standard for language-independent arithmetic (LIA-1).
A function type describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is \( T \), the function type is sometimes called “function returning \( T \)”. The construction of a function type from a return type is called “function type derivation”.

A pointer type may be derived from a function type or an object type, called the referenced type. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type \( T \) is sometimes called “pointer to \( T \)”. The construction of a pointer type from a referenced type is called “pointer type derivation”. A pointer type is a complete object type.

An atomic type describes the type designated by the construct `Atomic(type-name)`. (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

23 Arithmetic types and pointer types are collectively called scalar types. Array and structure types are collectively called aggregate types.\(^{50}\)

24 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

25 A type has known constant size if the type is not incomplete and is not a variable length array type.

26 Array, function, and pointer types are collectively called derived declarator types. A declarator type derivation from a type \( T \) is the construction of a derived declarator type from \( T \) by the application of an array-type, a function-type, or a pointer-type derivation to \( T \).

27 A type is characterized by its type category, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

28 Any type so far mentioned is an unqualified type. Each unqualified type has several qualified versions of its type,\(^{51}\) corresponding to the combinations of one, two, or all three of the const, volatile, and restrict qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.\(^{52}\) A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

29 Further, there is the _Atomic qualifier. The presence of the _Atomic qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase “atomic, qualified, or unqualified type” whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase “qualified or unqualified type”, without specific mention of atomic, does not include the atomic types.

30 A pointer to void shall have the same representation and alignment requirements as a pointer to a character type.\(^{52}\) Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

31 **EXAMPLE 1** The type designated as “float *” has type “pointer to float”. Its type category is pointer, not a floating type. The const-qualified version of this type is designated as “float const *” whereas the type designated as “const float *” is not a qualified type — its type is “pointer to const-qualified float” and is a pointer to a qualified type.

\(^{50}\)Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

\(^{51}\)See 6.7.3 regarding qualified array and function types.

\(^{52}\)The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.
EXAMPLE 2 The type designated as "struct tag *([5]) (float)" has type "array of pointer to function returning struct tag'. The array has length five and the function has a single parameter of type float. Its type category is array.

Forward references: compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause.

2 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

3 Values stored in unsigned bit-fields and objects of type unsigned char shall be represented using a pure binary notation.\(^{53}\)

4 Values stored in non-bit-field objects of any other object type consist of \( n \times \text{CHAR\_BIT} \) bits, where \( n \) is the size of an object of that type, in bytes. The value may be copied into an object of type unsigned char \([n]\) (e.g., by memcpy); the resulting set of bytes is called the object representation of the value. Values stored in bit-fields consist of \( m \) bits, where \( m \) is the size specified for the bit-field. The object representation is the set of \( m \) bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.\(^{54}\) Such a representation is called a trap representation.

6 When a value is stored in an object of an array, union or structure, each of which corresponds to an addressable storage unit, the bytes of the object representation that correspond to any padding bytes take unspecified values.\(^{55}\) The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.\(^{56}\) Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

9 Loads and stores of objects with atomic types are done with memory_order_seq_cst semantics.

Forward references: declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

6.2.6.2 Integer types

1 For unsigned integer types other than unsigned char, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are \( N \) value bits, each bit shall represent a different power of 2 between 1 and \( 2^{N-1} \), so that objects of that type shall be capable of representing values from 0 to \( 2^{N-1} - 1 \) using a pure binary representation;

\(^{53}\) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the American National Dictionary for Information Processing Systems.) A byte contains \( \text{CHAR\_BIT} \) bits, and the values of type unsigned char range from 0 to \( 2^{\text{CHAR\_BIT}} - 1 \).

\(^{54}\) Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

\(^{55}\) Thus, for example, structure assignment need not copy any padding bits.

\(^{56}\) It is possible for objects \( x \) and \( y \) with the same effective type \( T \) to have the same value when they are accessed as objects of type \( T \), but to have different values in other contexts. In particular, if \( == \) is defined for type \( T \), then \( x == y \) does not imply that \( \text{memcmp}(\&x, \&y, \text{sizeof}(T)) == 0 \). Furthermore, \( x == y \) does not necessarily imply that \( x \) and \( y \) have the same value; other operations on values of type \( T \) might distinguish between them.
this shall be known as the value representation. The values of any padding bits are unspecified.\textsuperscript{57}

For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; \texttt{signed char} shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are \(M\) value bits in the signed type and \(N\) in the unsigned type, then \(M \leq N\)). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

- the corresponding value with sign bit 0 is negated (sign and magnitude);
- the sign bit has the value \(-2^M\) (two’s complement);
- the sign bit has the value \(-2^M - 1\) (ones’ complement).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones’ complement), is a trap representation or a normal value. In the case of sign and magnitude and ones’ complement, if this representation is a normal value it is called a negative zero.

If the implementation supports negative zeros, they shall be generated only by:

- the \&, \|, ^, ^, <<, and >> operators with operands that produce such a value;
- the +, -, *, /, and \% operators where one operand is a negative zero and the result is zero;
- compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

If the implementation does not support negative zeros, the behavior of the \&, \|, ^, ^, <<, and >> operators with operands that would produce such a value is undefined.

The values of any padding bits are unspecified.\textsuperscript{58} A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The width of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

### 6.2.7 Compatible type and composite type

Two types have compatible type if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.\textsuperscript{59} Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of

\textsuperscript{57}Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

\textsuperscript{58}Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

\textsuperscript{59}Two types need not be identical to be compatible.
corresponding members are declared with compatible types; if one member of the pair is declared
with an alignment specifier, the other is declared with an equivalent alignment specifier; and if
one member of the pair is declared with a name, the other is declared with the same name. For
two structures, corresponding members shall be declared in the same order. For two structures or
unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding
members shall have the same values.

2 All declarations that refer to the same object or function shall have compatible type; otherwise, the
behavior is undefined.

3 A composite type can be constructed from two types that are compatible; it is a type that is compatible
with both of the two types and satisfies the following conditions:

   — If both types are array types, the following rules are applied:
     • If one type is an array of known constant size, the composite type is an array of that size.
     • Otherwise, if one type is a variable length array whose size is specified by an expression
       that is not evaluated, the behavior is undefined.
     • Otherwise, if one type is a variable length array whose size is specified, the composite
type is a variable length array of that size.
     • Otherwise, if one type is a variable length array of unspecified size, the composite type is
       a variable length array of unspecified size.
     • Otherwise, both types are arrays of unknown size and the composite type is an array of
       unknown size.

The element type of the composite type is the composite type of the two element types.

   — If only one type is a function type with a parameter type list (a function prototype), the
   composite type is a function prototype with the parameter type list.

   — If both types are function types with parameter type lists, the type of each parameter in the
   composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of
that identifier is visible, if the prior declaration specifies internal or external linkage, the type of
the identifier at the later declaration becomes the composite type.

Forward references: array declarators (6.7.6.2).

EXAMPLE Given the following two file scope declarations:

```c
int f(int (*)(int), double (*)(int)[3]);
int f(int (*)(char *), double (*)(int)[3]);
```

The resulting composite type for the function is:

```c
int f(int (*)(char *), double (*)(int)[3]);
```

### 6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at
which objects of that type may be allocated. An alignment is an implementation-defined integer
value representing the number of bytes between successive addresses at which a given object can be
allocated. An object type imposes an alignment requirement on every object of that type: stricter
alignment can be requested using the `_Alignas` keyword.

2 A fundamental alignment is a valid alignment less than or equal to `_Alignof (max_align_t)`. Funda-
mental alignments shall be supported by the implementation for objects of all storage durations. The
alignment requirements of the following types shall be fundamental alignments:

---

(As specified in 6.2.1, the later declaration might hide the prior declaration.)
— all atomic, qualified, or unqualified basic types;
— all atomic, qualified, or unqualified enumerated types;
— all atomic, qualified, or unqualified pointer types;
— all array types whose element type has a fundamental alignment requirement;
— all types specified in Clause 7 as complete object types;
— all structure or union types all of whose elements have types with fundamental alignment requirements and none of whose elements have an alignment specifier specifying an alignment that is not a fundamental alignment.

An extended alignment is represented by an alignment greater than \_Alignof((max_align_t)). It is implementation-defined whether any extended alignments are supported and the storage durations for which they are supported. A type having an extended alignment requirement is an over-aligned type.\(^{61}\)

Alignments are represented as values of the type size_t. Valid alignments include only fundamental alignments, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.

Alignments have an order from weaker to stronger or stricter alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

The alignment requirement of a complete type can be queried using an \_Alignof expression. The types char, signed char, and unsigned char shall have the weakest alignment requirement.

Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.
— Two alignments are different when their numeric values are not equal.
— When an alignment is larger than another it represents a stricter alignment.

### 6.3 Conversions

Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an implicit conversion, as well as those that result from a cast operation (an explicit conversion). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.

Unless explicitly stated otherwise, conversion of an operand value to a compatible type causes no change to the value or the representation.

**Forward references:** cast operators (6.5.4).

#### 6.3.1 Arithmetic operands

**6.3.1.1 Boolean, characters, and integers**

Every integer type has an integer conversion rank defined as follows:

— No two signed integer types shall have the same rank, even if they have the same representation.
— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

\(^{61}\)Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.
— The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

— The rank of `char` shall equal the rank of `signed char` and `unsigned char`.

— The rank of `_Bool` shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

2 The following may be used in an expression wherever an `int` or `unsigned int` may be used:

— An object or expression with an integer type (other than `int` or `unsigned int`) whose integer conversion rank is less than or equal to the rank of `int` and `unsigned int`.

— A bit-field of type `_Bool`, `int`, `signed int`, or `unsigned int`.

If an `int` can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the integer promotions. All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a “plain” `char` can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

6.3.1.2 Boolean type

1 When any scalar value is converted to `_Bool`, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

---

62) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary `+`, `-`, and `~` operators, and to both operands of the shift operators, as specified by their respective subclauses.

63) NaNs do not compare equal to 0 and thus convert to 1.

64) The rules describe arithmetic on the mathematical value, not the value of a given type of expression.
6.3.1.4 Real floating and integer

1 When a finite value of real-standard floating type is converted to an integer type other than _Bool, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.\(^{65}\)

2 When a finite value of decimal floating type is converted to an integer type other than _Bool, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the “invalid” floating-point exception shall be raised and the result of the conversion is unspecified.

3 When a value of integer type is converted to a real-standard floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined. Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

4 When a value of integer type is converted to a decimal floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEC 60559.

6.3.1.5 Real floating types

1 When a value of real floating type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged.\(^{66}\)

2 When a value of real floating type is converted to a standard floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.

3 When a value of real floating type is converted to a decimal floating type, if the value being converted cannot be represented exactly, the result is correctly rounded with exceptions raised as specified in IEC 60559.

4 Results of some implicit conversions may be represented in greater range and precision than that required by the new type (see 6.3.1.8 and 6.8.6.4).

6.3.1.6 Complex types

1 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

6.3.1.7 Real and complex

1 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.

2 When a value of complex type is converted to a real type other than _Bool,\(^{66}\) the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

6.3.1.8 Usual arithmetic conversions

1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose

\(^{65}\) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. Thus, the range of portable real floating values is \(-1 \ll \text{type}_\text{MAX} + 1\).

\(^{66}\) See 6.3.1.2.
The values of floating operands and of the results of floating expressions may be represented in

An "rvalue" is in this document described as the "value of an expression".

double _Complex

§ 6.3.2.1 Language

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

An lvalue is an expression (with an object type other than void) that potentially designates an object; if an lvalue does not designate an object when it is evaluated, the behavior is undefined.

67) For example, addition of a double _Complex and a float entails just the conversion of the float operand to double (and yields a double _Complex result).

68) The name "lvalue" comes originally from the assignment expression E1 = E2, in which the left operand E1 is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object "locator value". What is sometimes called "rvalue" is in this document described as the "value of an expression".

An obvious example of an lvalue is an identifier of an object. As a further example, if E is a unary expression that is a pointer to an object, *E is an lvalue that designates the object to which E points.

\[ \text{If one operand has decimal floating type, the other operand shall not have standard floating, complex, or imaginary type.} \]

First, if the type of either operand is _Decimal128, the other operand is converted to _Decimal128.

Otherwise, if the type of either operand is _Decimal64, the other operand is converted to _Decimal64.

Otherwise, if the type of either operand is _Decimal32, the other operand is converted to _Decimal32.

Otherwise, if the corresponding real type of either operand is long double, the other operand is converted, without change of type domain, to a type whose corresponding real type is long double.

Otherwise, if the corresponding real type of either operand is double, the other operand is converted, without change of type domain, to a type whose corresponding real type is double.

Otherwise, if the corresponding real type of either operand is float, the other operand is converted, without change of type domain, to a type whose corresponding real type is float.\(^67\)

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

If both operands have the same type, then no further conversion is needed.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

The values of floating operands and of the results of floating expressions may be represented in greater range and precision than that required by the type; the types are not changed thereby. See 5.2.4.2.2 regarding evaluation formats.

6.3.2 Other operands
When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

Except when it is the operand of the `sizeof` operator, the unary `&` operator, the `++` operator, the `--` operator, or the left operand of the . operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue); this is called *lvalue conversion*. If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; additionally, if the lvalue has atomic type, the value has the non-atomic version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined. If the lvalue designates an object of automatic storage duration that could have been declared with the `register` storage class (never had its address taken), and that object is uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

Except when it is the operand of the `sizeof` operator, or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type "array of type" is converted to an expression with type "pointer to type" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

A *function designator* is an expression that has function type. Except when it is the operand of the `sizeof` operator, or the unary `&` operator, a function designator with type "function returning type" is converted to an expression that has type "pointer to function returning type".

Forward references: address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions `<stddef.h>` (7.19), initialization (6.7.9), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the `sizeof` and `_Alignof` operators (6.5.3.4), structure and union members (6.5.2.3).

6.3.2.2 **void**

1 The (nonexistent) value of a *void expression* (an expression that has type `void`) shall not be used in any way, and implicit or explicit conversions (except to `void`) shall not be applied to such an expression. If an expression of any other type is evaluated as a void expression, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.3.2.3 **Pointers**

1 A pointer to `void` may be converted to or from a pointer to any object type. A pointer to any object type may be converted to a pointer to `void` and back again; the result shall compare equal to the original pointer.

2 For any qualifier `q`, a pointer to a non-`q`-qualified type may be converted to a pointer to the `q`-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

3 An integer constant expression with the value 0, or such an expression cast to type `void *`, is called a *null pointer constant*. If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.

6 Any pointer type may be converted to an integer type. Except as previously specified, the result

---

69) Because this conversion does not occur, the operand of the `sizeof` operator remains a function designator and violates the constraints in 6.5.3.4.

70) The macro `NULL` is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.19.

71) The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.
is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.

7   A pointer to an object type may be converted to a pointer to a different object type. If the resulting pointer is not correctly aligned\(^\text{72}\) for the referenced type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

8   A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined.

Forward references: cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.20.1.4), simple assignment (6.5.16.1).

\(^{72}\)In general, the concept “correctly aligned” is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.
6.4 Lexical elements

Syntax

1 token:
   keyword
   identifier
   constant
   string-literal
   punctuator

preprocessing-token:
   header-name
   identifier
   pp-number
   character-constant
   string-literal
   punctuator
   each non-white-space character that cannot be one of the above

Constraints

2 Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, or a punctuator.

Semantics

3 A token is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.\(^{73}\) If a ‘ ‘ or a ” character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (described later), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

4 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. There is one exception to this rule: header name preprocessing tokens are recognized only within \#include preprocessing directives and in implementation-defined locations within \#pragma directives. In such contexts, a sequence of characters that could be either a header name or a string literal is recognized as the former.

5 EXAMPLE 1 The program fragment 1Ex is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens 1 and Ex might produce a valid expression (for example, if Ex were a macro defined as +1). Similarly, the program fragment 1E1 is parsed as a preprocessing number (one that is a valid floating constant token), whether or not E is a macro name.

6 EXAMPLE 2 The program fragment x+++++y is parsed as x ++ ++ + y, which violates a constraint on increment operators, even though the parse x ++ + ++ y might yield a correct expression.

Forward references: character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5–6.4.5).

\(^{73}\) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files.
6.4.1 Keywords

Syntax

```
keyword: one of
  auto  signed  case  static
  break  sizeof  char  struct
  case  static  const  switch
  char  struct  continue  typedef
  const  switch  default  union
  continue  typedef  do  unsigned
  default  union  double  void
  do  unsigned  else  volatile
  double  void  enum  while
  else  volatile  extern  _Alignas
  enum  while  float  _Alignof
  extern  _Alignas  for  _Atomic
  float  _Alignof  goto  _Bool
  for  _Atomic  if  _Complex
  goto  _Bool  inline  _Decimal128
  if  _Complex  int  _Decimal32
  inline  _Generic  long  _Decimal64
  int  _Imaginary  register  _Generic
  long  _Noreturn  restrict  _Imaginary
  register  _Static_assert  return  _Noreturn
  restrict  _Thread_local  short  _Static_assert
  return  auto  signed  _Thread_local
  short  break  sizeof
```

Semantics

The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords except in an attribute token, and shall not be used otherwise. The keyword _Imaginary is reserved for specifying imaginary types.\(^{74}\)

6.4.2 Identifiers

6.4.2.1 General

Syntax

```
identifier: identifier-nondigit
  identifier  identifier-nondigit
  identifier  digit

identifier-nondigit:
  nondigit
  universal-character-name
  other implementation-defined characters

nondigit: one of
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z
```

\(^{74}\)One possible specification for imaginary types appears in Annex G.
digit: one of

0 1 2 3 4 5 6 7 8 9

Semantics

An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.

The use of universal character names in identifiers is specified in Annex D: Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in D.1.\(^75\) The initial character shall not be a universal character name designating a character whose encoding falls into one of the ranges specified in D.2. An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation-defined.

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword except in an attribute token.

Implementation limits

As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an external name (an identifier that has external linkage) may be more restrictive than that for an internal name (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.

Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

Forward references: universal character names (6.4.3), macro replacement (6.10.3).

6.4.2.2 Predefined identifiers

Semantics

The identifier __func__ shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[] = "function-name";
```

appeared, where function-name is the name of the lexically-enclosing function.\(^76\)

This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5.

EXAMPLE Consider the code fragment:

```c
#define <stdio.h>
void myfunc(void)
{
  printf("%s\n", __func__);  
  /* ... */
}
```

Each time the function is called, it will print to the standard output stream:

\(^75\)On systems in which linkers cannot accept extended characters, an encoding of the universal character name can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the \(\cup\) in a universal character name. Extended characters can produce a long external identifier.

\(^76\)Since the name __func__ is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name __func__, the behavior is undefined.
Forward references: function definitions (6.9.1).

6.4.3 Universal character names

Syntax

universal-character-name:
  \u hex-quad
  \U hex-quad hex-quad

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

Constraints

A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (\$), 0040 (@), or 0060 (‘), nor one in the range D800 through DFFF inclusive.

Description

Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

Semantics

The universal character name \U{nnnnnnnn} designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is \u{nnnnnn}. Similarly, the universal character name \u{nnnn} designates the character whose four-digit short identifier is \n{nnnn} (and whose eight-digit short identifier is 0000nnnn).

---

77) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF–16).

78) Short identifiers for characters were first specified in ISO/IEC 10646–1:1993/Amd 9:1997.
6.4.4 Constants

Syntax

constant:
  integer-constant
  floating-constant
  enumeration-constant
  character-constant

Constraints

Each constant shall have a type and the value of a constant shall be in the range of representable values for its type.

Semantics

Each constant has a type, determined by its form and value, as detailed later.

6.4.4.1 Integer constants

Syntax

integer-constant:
  decimal-constant integer-suffix_opt
  octal-constant integer-suffix_opt
  hexadecimal-constant integer-suffix_opt

decimal-constant:
  nonzero-digit
  decimal-constant digit

octal-constant:
  0
  octal-constant octal-digit

hexadecimal-constant:
  hexadecimal-prefix hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of
  -0x 0x 0X

nonzero-digit: one of
  1 2 3 4 5 6 7 8 9

octal-digit: one of
  0 1 2 3 4 5 6 7

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffix_opt
  unsigned-suffix long-long-suffix
  long-suffix unsigned-suffix_opt
  long-long-suffix unsigned-suffix_opt

52 Language § 6.4.4.1
Description

2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

3 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix `0` optionally followed by a sequence of the digits `0` through `7` only. A hexadecimal constant consists of the prefix `0x` or `0X` followed by a sequence of the decimal digits and the letters `a` (or `A`) through `f` (or `F`) with values `10` through `15` respectively.

Semantics

4 The value of a decimal constant is computed base `10`; that of an octal constant, base `8`; that of a hexadecimal constant, base `16`. The lexically first digit is the most significant.

5 The type of an integer constant is the first of the corresponding list in which its value can be represented.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal Constant</th>
<th>Octal or Hexadecimal Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>long long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ll or LL</td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long int</td>
</tr>
</tbody>
</table>

6 If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. If an integer constant cannot be represented by any type in its list and has no extended integer type, then the integer constant has no type.

Forward references: preprocessing numbers (6.4.8), numeric conversion functions (7.22.1).
6.4.4.2 Floating constants

Syntax

1 floating-constant:
   decimal-floating-constant
   hexadecimal-floating-constant

   decimal-floating-constant:
      fractional-constant exponent-part\textsubscript{opt} floating-suffix\textsubscript{opt}
      digit-sequence exponent-part floating-suffix\textsubscript{opt}

   hexadecimal-floating-constant:
      hexadecimal-prefix hexadecimal-fractional-constant
      binary-exponent-part floating-suffix\textsubscript{opt}
      hexadecimal-prefix hexadecimal-digit-sequence
      binary-exponent-part floating-suffix\textsubscript{opt}

   fractional-constant:
      digit-sequence\textsubscript{opt} . digit-sequence
      digit-sequence .

   exponent-part:
      e sign\textsubscript{opt} digit-sequence
      E sign\textsubscript{opt} digit-sequence

   sign: one of
      + -

   digit-sequence:
      digit
      digit-sequence digit

   hexadecimal-fractional-constant:
      hexadecimal-digit-sequence\textsubscript{opt} .
      hexadecimal-digit-sequence hexadecimal-digit-sequence
      hexadecimal-digit-sequence .

   binary-exponent-part:
      p sign\textsubscript{opt} digit-sequence
      P sign\textsubscript{opt} digit-sequence

   hexadecimal-digit-sequence:
      hexadecimal-digit
      hexadecimal-digit-sequence hexadecimal-digit

   floating-suffix: one of
      f l F L df dd dl DF DD DL

Constraints

2 A floating suffix \texttt{df}, \texttt{dd}, \texttt{dl}, \texttt{DF}, \texttt{DD}, or \texttt{DL} shall not be used in a hexadecimal floating constant.

Description

3 A floating constant has a significand part that may be followed by an exponent part and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. The components of the exponent part are an \texttt{e}, \texttt{E}, \texttt{p}, or \texttt{P} followed by an exponent
consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

**Semantics**

4 The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when FLT_RADIX is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when FLT_RADIX is a power of 2, the result is correctly rounded.

5 An unsuffixed floating constant has type `double`. If suffixed by the letter for a floating suffix it has a type according to the following table: if suffixed by the letter F or f, it has type.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>f, F</td>
<td>float</td>
</tr>
<tr>
<td>l, L</td>
<td>long double</td>
</tr>
<tr>
<td>dl, DL</td>
<td>_Decimal128</td>
</tr>
<tr>
<td>dd, DD</td>
<td>_Decimal64</td>
</tr>
</tbody>
</table>

6 The values of floating constants may be represented in greater range and precision than that required by the type (determined by the suffix); the types are not changed thereby. See 5.2.4.2.2 regarding evaluation formats.

7 Floating constants of decimal floating type that have the same numerical value but different quantum exponents have distinguishable internal representations. The value shall be correctly rounded as specified in IEC 60559. The coefficient $c$ and the quantum exponent $q$ of a finite converted decimal floating-point number (see 5.2.4.2.3) are determined as follows:

- $q$ is set to the value of $\text{sign}\_\text{opt}$ digit-sequence in the exponent part, if any, or to 0, otherwise.
- If there is a fractional constant, $q$ is decreased by the number of digits to the right of the period and the period is removed to form a digit sequence.
- $c$ is set to the value of the digit sequence (after any period has been removed).
- Rounding required because of insufficient precision or range in the type of the result will round $c$ to the full precision available in the type, and will adjust $q$ accordingly within the limits of the type, provided the rounding does not yield an infinity (in which case the result is an appropriately signed internal representation of infinity). If the full precision of the type would require $q$ to be smaller than the minimum for the type, then $q$ is pinned at the minimum and $c$ is adjusted through the subnormal range accordingly, perhaps to zero.

8 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution.
time. All floating constants of the same source form\textsuperscript{80) shall convert to the same internal format with the same value.

\textbf{EXAMPLE} Following are floating constants of type \_Decimal64 and their values as triples \((s, c, q)\). Note that for \_Decimal64, the precision (maximum coefficient length) is 16 and the quantum exponent range is \(-398 \leq q \leq 369\).

\begin{verbatim}
8.00 dd (1.0, 0)
9.000 dd (1.0, -2)
123. dd (1.23, 0)
1.23E3dd (1.23, 1)
1.23E+7dd (1.23, 6)
12. 0dd (1.20, -1)
12. 3dd (1.23, -1)
8.00123dd (1.23, -5)
1.23e-12dd (1.23, -14)
1234.9E-4dd (1.2345, -3)
9E+7dd (1, 0, 7)
12345678901234567890 dd (1.2345678901234567890, 4) assuming default rounding and DEC_EVAL_METHOD is 0 or 1\textsuperscript{81)}
1234E-406dd (1.12, -398) assuming default rounding and DEC_EVAL_METHOD is 0 or 1
1234E-402dd (1.0, -398) assuming default rounding and DEC_EVAL_METHOD is 0 or 1
10000. dd (1, 10000, 0)
._00010dd (1, 1, -4)
1000.0dd (1, 1000, 0)
._0001e00dd (1, 1, -4)
1000.0dd (1, 100000, 0)
8.0001dd (1, 1, -4)
100000.0dd (1, 100000, 0)
80.0001dd (1, 1, -4)
801000. dd (1, 100000, 0)
801000.0dd (1, 100000, 0)
801000.00dd (1, 100000, 0)
80.000dd (1, 1, -2)
90. dd (1, 0, 0)
._000dd (1, 0, -2)
80.00e-5dd (1, 0, -7)
80. e-5dd (1, 0, -5)
._00e-5dd (1, 0, -7)
\end{verbatim}

\textbf{Recommended practice}

The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.

The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as \texttt{strtod}, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding\textsuperscript{82)}.

\textbf{Forward references:} preprocessing numbers (6.4.8), numeric conversion functions (7.22.1).

\textbf{6.4.4.3 Enumeration constants}

\textbf{Syntax}

\begin{verbatim}
enumeration-constant:
identifier
\end{verbatim}

\textsuperscript{80)} 1.23, 1.230, 123e-2, 123e-02, and 1.23L are all different source forms and thus need not convert to the same internal format and value.

\textsuperscript{81)} That is, assuming the default translation rounding-direction mode is not changed by an \texttt{FE朱 DEC ROUND} pragma (7.6.3).

\textsuperscript{82)} The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.22.1.5).
An identifier declared as an enumeration constant has type int.

Forward references: enumeration specifiers (6.7.2.2).

## 6.4.4.4 Character constants

### Syntax

character-constant:

- c-char-sequence encoding-prefix opt c-char-sequence

encoding-prefix:

- L c-char-sequence u8
- U c-char-sequence u

- c-char-sequence:
  - c-char
  - c-char-sequence c-char

- c-char:
  - any member of the source character set except the single-quote ‘, backslash \, or new-line character escape-sequence

- escape-sequence:
  - simple-escape-sequence
  - octal-escape-sequence
  - hexadecimal-escape-sequence
  - universal-character-name

- simple-escape-sequence: one of \ a b f n r t v

- octal-escape-sequence:
  - \ octal-digit
  - \ octal-digit octal-digit
  - \ octal-digit octal-digit octal-digit

- hexadecimal-escape-sequence:
  - \x hexadecimal-digit
  - \x hexadecimal-escape-sequence hexadecimal-digit

## Description

An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in ‘x’. A UTF-8 character constant is the same, except prefixed by u8. A wide character constant is the same, except prefixed by the letter L, U, or U. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

The single-quote ‘, the double-quote ”, the question-mark ?, the backslash \, and arbitrary integer values are representable according to the following table of escape sequences:
4 The double-quote " and question-mark ? are representable either by themselves or by the escape sequences \" and \?, respectively, but the single-quote ' and the backslash \ shall be represented, respectively, by the escape sequences \' and \\.

5 The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

6 The hexadecimal digits that follow the backslash and the letter x in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: \a, \b, \f, \n, \r, \t, and \v.\(^{83}\)

Constraints

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Corresponding Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>unsigned char</td>
</tr>
<tr>
<td>u8</td>
<td>unsigned char</td>
</tr>
<tr>
<td>L</td>
<td>the unsigned type corresponding to wchar_t</td>
</tr>
<tr>
<td>u</td>
<td>char16_t</td>
</tr>
<tr>
<td>U</td>
<td>char32_t</td>
</tr>
</tbody>
</table>

10 A UTF-8 character constant shall not contain more than one character.\(^{84}\) The value shall be representable with a single UTF-8 code unit.

Semantics

11 An integer character constant has type int. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., ‘ab’), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type char whose value is that of the single character or escape sequence is converted to type int.

12 A UTF-8 character constant has type unsigned char. The value of a UTF-8 character constant is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-8 code unit.

13 A wide character constant prefixed by the letter L has type wchar_t, an integer type defined in the <stddef.h> header; a wide character constant prefixed by the letter u or U has type char16_t or char32_t, respectively, unsigned integer types defined in the <wchar.h> header. The value of a

\(^{83}\) The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See “future language directions” (6.11.4).

\(^{84}\) For example u8 ‘ab’ violates this constraint.
wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the \texttt{mbtowc}, \texttt{mbtocr16}, or \texttt{mbtocr32} function as appropriate for its type, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

\textbf{EXAMPLE 1} The construction \texttt{\0} is commonly used to represent the null character.

\textbf{EXAMPLE 2} Consider implementations that use two’s complement representation for integers and eight bits for objects that have type \texttt{char}. In an implementation in which type \texttt{char} has the same range of values as \texttt{signed char}, the integer character constant \texttt{\xFF} has the value $-1$; if type \texttt{char} has the same range of values as \texttt{unsigned char}, the character constant \texttt{\xFF} has the value $+255$.

\textbf{EXAMPLE 3} Even if eight bits are used for objects that have type \texttt{char}, the construction \texttt{\x123} specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are \texttt'{\x12}’ and \texttt{'3’}, the construction \texttt{'\0223'} can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

\textbf{EXAMPLE 4} Even if 12 or more bits are used for objects that have type \texttt{wchar_t}, the construction \texttt{L'\1234'} specifies the implementation-defined value that results from the combination of the values \texttt{'0123'} and \texttt{'4’}.

\textbf{Forward references:} common definitions \texttt{<stddef.h>} (7.19), the \texttt{mbtowc} function (7.22.7.2), Unicode utilities \texttt{<uchar.h>} (7.28).

\section*{6.4.5 String literals}

\textbf{Syntax}

\begin{verbatim}
string-literal: encoding-prefix_opt " s-char-sequence_opt " encoding-prefix:
    u8
    u
    U
    L
    " s-char-sequence_opt "

s-char-sequence:
    s-char
    s-char-sequence s-char

s-char: any member of the source character set except
    the double-quote \"\, backslash \", or new-line character escape-sequence
\end{verbatim}

\textbf{Constraints}

A sequence of adjacent string literal tokens shall not include both a wide string literal and a UTF–8 string literal.

\textbf{Description}

A \textit{character string literal} is a sequence of zero or more multibyte characters enclosed in double-quotes, as in \texttt{"xyz"}. A UTF–8 \textit{string literal} is the same, except prefixed by \texttt{u8}. A \textit{wide string literal} is the same, except prefixed by the letter \texttt{L}, \texttt{u}, or \texttt{U}.

The same considerations apply to each element of the sequence in a string literal as if it were in an integer character constant (for a character or UTF–8 string literal) or a wide character constant (for a wide string literal), except that the single-quote ‘ is representable either by itself or by the escape sequence \texttt{\'}, but the double-quote " shall be represented by the escape sequence \texttt{\"}.
Semantics

5 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and identically-prefixed string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens has an encoding prefix, the resulting multibyte character sequence is treated as having the same prefix; otherwise, it is treated as a character string literal. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence are implementation-defined.

6 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals. The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type char, and are initialized with the individual bytes of the multibyte character sequence. For UTF-8 string literals, the array elements have type char, and are initialized with the characters of the multibyte character sequence, as encoded in UTF-8. For wide string literals prefixed by the letter L, the array elements have type wchar_t and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the mbstowcs function with an implementation-defined current locale. For wide string literals prefixed by the letter u or U, the array elements have type char16_t or char32_t, respectively, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by successive calls to the mbrtoc16, or mbrtoc32 function as appropriate for its type, with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

7 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

8 EXAMPLE 1 This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are ‘\x12’ and ‘3’, because escape sequences are converted into single members of the execution character set just prior to adjacent string literal concatenation.

9 EXAMPLE 2 Each of the sequences of adjacent string literal tokens

```
"a" "b" L"c"
"a" L"b" "c"
L"a" "b" L"c"
L"a" L"b" L"c"
```

is equivalent to the string literal

```
L"abc"
```

Likewise, each of the sequences

```
"a" "b" u"c"
"a" u"b" "c"
u"a" "b" u"c"
u"a" u"b" u"c"
```

is equivalent to

```
u"abc"
```

Forward references: common definitions <stddef.h> (7.19), the mbstowcs function (7.22.8.1), Unicode utilities <uchar.h> (7.28).

85) A string literal might not be a string (see 7.1.1), because a null character can be embedded in it by a \0 escape sequence.
6.4.6 Punctuators

Syntax

1. punctuator: one of
   - [ ] ( ) { } -> ++ -- & * + - ~ ! / % << >> <

Semantics

2. A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an operator (other forms of operator also exist in some contexts). An operand is an entity on which an operator acts.

3. In all aspects of the language, the six tokens behave, respectively, the same as the six tokens except for their spelling.

Forward references: expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8).

6.4.7 Header names

Syntax

1. header-name: < h-char-sequence > " q-char-sequence "

   h-char-sequence: h-char
                   h-char-sequence h-char

   h-char: any member of the source character set except the new-line character and >

   q-char-sequence: q-char
                   q-char-sequence q-char

   q-char: any member of the source character set except the new-line character and "

86) These tokens are sometimes called “digraphs”.
87) Thus [ and <: behave differently when “stringized” (see 6.10.3.2), but can otherwise be freely interchanged.
Semantics

The sequences in both forms of header names are mapped in an implementation-defined manner to
headers or external source file names as specified in 6.10.2.

If the characters \', \", \\, or \\/* occur in the sequence between the < and > delimiters, the behavior
is undefined. Similarly, if the characters \', \\, \\/, or \\/* occur in the sequence between the "
delimiters, the behavior is undefined.\(^{88}\) Header name preprocessing tokens are recognized only
within \#include preprocessing directives and in implementation-defined locations within \#pragma
directives.\(^{89}\)

EXAMPLE The following sequence of characters:

\[
0x3<1/a.h>1e2
\#include <1/a.h>
\#define const member@$
\]

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a / on the left
and a \ on the right).

\[
{0x3}\{<\}\{1\}\{/\}\{1\}\{.\}\{a\}\{.\}\{h\}\{>\}\{1\}\{e\}\{2\}
\{#\}\{include\} \{<1/a.h>\}
\{#\}\{define\} \{const\}\{.\}\{member\}\{@\}\{$
\]

Forward references: source file inclusion (6.10.2).

6.4.8 Preprocessing numbers

Syntax

1  pp-number:

   \[\begin{align*}
   &\text{digit} \\
   &\quad \rightarrow \text{digit} \\
   &\quad \text{pp-number \ digit} \\
   &\quad \text{pp-number \ identifier-nondigit} \\
   &\quad \text{pp-number \ e \ sign} \\
   &\quad \text{pp-number \ E \ sign} \\
   &\quad \text{pp-number \ p \ sign} \\
   &\quad \text{pp-number \ P \ sign} \\
   &\quad \text{pp-number \ \rightarrow \rightarrow}
\end{align*}\]

Description

1  A preprocessing number begins with a digit optionally preceded by a period (.) and may be followed
by valid identifier characters and the character sequences e+, e-, E+, E-, p+, p-, P+, or P-.

2  Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

1  A preprocessing number does not have type or a value; it acquires both after a successful conversion
(as part of translation phase 7) to a floating constant token or an integer constant token.

6.4.9 Comments

1  Except within a character constant, a string literal, or a comment, the characters \\/* introduce a
comment. The contents of such a comment are examined only to identify multibyte characters and
to find the characters */ that terminate it.\(^{90}\)

---

\(^{88}\)Thus, sequences of characters that resemble escape sequences cause undefined behavior.

\(^{89}\)For an example of a header name preprocessing token used in a \#pragma directive, see 6.10.9.

\(^{90}\)Thus, \\/* \ldots */ comments do not nest.
Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

**EXAMPLE**

```
"a//b"            // four-character string literal
#include "//e"    // undefined behavior
// */              // comment, not syntax error
f = g/**/h;      // equivalent to f = g / h;
//\               // comment, not syntax error
i();             // part of a two-line comment
/\                // part of a two-line comment
#define glue(x,y) x##y
/\                  // syntax error, not comment
glue(//,) k();     // equivalent to l();
/\/*/*/+ p;        // equivalent to m = n + p;
```
6.5 Expressions

1 An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. The value computations of the operands of an operator are sequenced before the value computation of the result of the operator.

2 If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. If there are multiple allowable orderings of the subexpressions of an expression, the behavior is undefined if such an unsequenced side effect occurs in any of the orderings.

3 The grouping of operators and operands is indicated by the syntax.

4 Some operators (the unary operator \(\sim\), and the binary operators \(<\ll, \gg, \&\&, \^\), and |, collectively described as bitwise operators) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.

5 If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

6 The effective type of an object for an access to its stored value is the declared type of the object, if any. If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using memcpy or memmove, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

— a type compatible with the effective type of the object,

---

\[i = ++i + 1;\]
\[a[i++] = i;\]

while allowing

\[i = i + 1;\]
\[a[i] = i;\]

---

\[i = i + 1;\]
\[a[i] = i;\]

---

Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

This paragraph renders undefined statement expressions such as

\[i = i + 1;\]
\[a[i] = i;\]

---

The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses ( ) (6.5.1), subscripting brackets [ ] (6.5.2.1), function-call parentheses ( ) (6.5.2.2), and the conditional operator ?: (6.5.15).

Within each major subclause, the operators have the same precedence. Left- or right-associativity is indicated in each subclause by the syntax for the expressions discussed therein.

In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations.

Allocated objects have no declared type.

The intent of this list is to specify those circumstances in which an object can or cannot be aliased.
— a qualified version of a type compatible with the effective type of the object,
— a type that is the signed or unsigned type corresponding to the effective type of the object,
— a type that is the signed or unsigned type corresponding to a qualified version of the effective
type of the object,
— an aggregate or union type that includes one of the aforementioned types among its members
(including, recursively, a member of a subaggregate or contained union), or
— a character type.

A floating expression may be contracted, that is, evaluated as though it were a single opera-
tion, thereby omitting rounding errors implied by the source code and the expression evaluation
method. The FP_CONTRACT pragma in <math.h> provides a way to disallow contracted ex-
pressions. Otherwise, whether and how expressions are contracted is implementation-defined.

Operators involving decimal floating types are evaluated according to the semantics of IEC 60559,
including production of results with the preferred quantum exponent as specified in IEC 60559.

Forward references: the FP_CONTRACT pragma (7.12.2), copying functions (7.24.2).

6.5.1 Primary expressions

Syntax
1

primary-expression: identifier
constant
string-literal
( expression )
generic-selection

Semantics
2
An identifier is a primary expression, provided it has been declared as designating an object (in
which case it is an lvalue) or a function (in which case it is a function designator).

3
A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.

4
A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.

5
A parenthesized expression is a primary expression. Its type and value are identical to those of
the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the
unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

6
A generic selection is a primary expression. Its type and value depend on the selected generic
association, as detailed in the following subclause.

Forward references: declarations (6.7).

6.5.1.1 Generic selection

Syntax
1

generic-selection: __Generic ( assignment-expression , generic-assoc-list )

97) The intermediate operations in the contracted expression are evaluated as if to infinite range and precision, while the
final operation is rounded to the format determined by the expression evaluation method. A contracted expression might
also omit the raising of floating-point exceptions.

98) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple
C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions,
their use needs to be well-defined and clearly documented.

99) Thus, an undeclared identifier is a violation of the syntax.
generic-assoc-list:
  generic-association
generic-association:
  type-name : assignment-expression
default : assignment-expression

Constraints
2 A generic selection shall have no more than one default generic association. The type name in a
generic association shall specify a complete object type other than a variably modified type. No two
generic associations in the same generic selection shall specify compatible types. The type of the
controlling expression is the type of the expression as if it had undergone an lvalue conversion,\footnote{An lvalue conversion drops type qualifiers.} array to pointer conversion, or function to pointer conversion. That type shall be compatible with at
most one of the types named in the generic association list. If a generic selection has no default
generic association, its controlling expression shall have type compatible with exactly one of the
types named in its generic association list.

Semantics
3 The controlling expression of a generic selection is not evaluated. If a generic selection has a generic
association with a type name that is compatible with the type of the controlling expression, then the
result expression of the generic selection is the expression in that generic association. Otherwise, the
result expression of the generic selection is the expression in the default generic association. None
of the expressions from any other generic association of the generic selection is evaluated.
4 The type and value of a generic selection are identical to those of its result expression. It is an
lvalue, a function designator, or a void expression if its result expression is, respectively, an lvalue, a
function designator, or a void expression.

EXAMPLE The \texttt{cbrt} type-generic macro could be implemented as follows:

\begin{verbatim}
#define cbrt(X) _Generic((X),
   long double: cbtrl, 
   default: cbrt, 
   float: cbrtf 
   ) (X)
\end{verbatim}

See 7.25 how such a macro could be implemented with the required rounding properties.

6.5.2 Postfix operators

Syntax
1 postfix-expression:
   primary-expression
   postfix-expression [ expression ]
   postfix-expression ( argument-expression-list_opt )
   postfix-expression . identifier
   postfix-expression -> identifier
   postfix-expression ++
   postfix-expression --
   ( type-name ) { initializer-list }
   ( type-name ) { initializer-list , }

argument-expression-list:
   assignment-expression
   argument-expression-list , assignment-expression

\footnote{An lvalue conversion drops type qualifiers.}
6.5.2.1 Array subscripting

Constraints
1. One of the expressions shall have type “pointer to complete object type”, the other expression shall have integer type, and the result has type “type”.

Semantics
2. A postfix expression followed by an expression in square brackets [ ] is a subscripted designation of an element of an array object. The definition of the subscript operator [ ] is that \( E1[E2] \) is identical to \( *((E1)+(E2)) \). Because of the conversion rules that apply to the binary + operator, if \( E1 \) is an array object (equivalently, a pointer to the initial element of an array object) and \( E2 \) is an integer, \( E1[E2] \) designates the \( E2 \)-th element of \( E1 \) (counting from zero).

3. Successive subscript operators designate an element of a multidimensional array object. If \( E \) is an \( n \)-dimensional array (\( n \geq 2 \)) with dimensions \( i \times j \times \cdots \times k \), then \( E \) (used as other than an lvalue) is converted to a pointer to an \((n - 1)\)-dimensional array with dimensions \( j \times \cdots \times k \). If the unary * operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the referenced \((n - 1)\)-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

4. EXAMPLE Consider the array object defined by the declaration

\[
\text{int } x[3][5];
\]

Here \( x \) is a \( 3 \times 5 \) array of int s; more precisely, \( x \) is an array of three element objects, each of which is an array of five int s. In the expression \( x[1] \), which is equivalent to \( *((x)+(1)) \), \( x \) is first converted to a pointer to the initial array of five int s. Then \( i \) is adjusted according to the type of \( x \), which conceptually entails multiplying \( i \) by the size of the object to which the pointer points, namely an array of five int objects. The results are added and indirection is applied to yield an array of five int s. When used in the expression \( x[1][j] \), that array is in turn converted to a pointer to the first of the int s, so \( x[1][j] \) yields an int.

Forward references: additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.6.2).

6.5.2.2 Function calls

Constraints
1. The expression that denotes the called function\(^{101}\) shall have type pointer to function returning \textbf{void} or returning a complete object type other than an array type.

2. If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics
3. A postfix expression followed by parentheses ( ) containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

4. An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument\(^{102}\).

5. If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type \textbf{void}.

\(^{101}\)Most often, this is the result of converting an identifier that is a function designator.

\(^{102}\)A function can change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function can then change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.
If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type `float` are promoted to `double`. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`..., ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined.

If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.

No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

There is a sequence point after the evaluations of the function designator and the actual arguments but before the actual call. Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.\(^\text{103}\)

Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

**EXAMPLE** In the function call

\[
(*pf[f1()]) (f2(), f3() + f4())
\]

the functions `f1`, `f2`, `f3`, and `f4` can be called in any order. All side effects have to be completed before the function pointed to by `pf[f1()]` is called.

---

**Forward references:** function declarators (including prototypes) (6.7.6.3), function definitions (6.9.1), the `return` statement (6.8.6.4), simple assignment (6.5.16.1).

### 6.5.2.3 Structure and union members

**Constraints**

1. The first operand of the `. ` operator shall have an atomic, qualified, or unqualified structure or union type, and the second operand shall name a member of that type.

2. The first operand of the `-> ` operator shall have type “pointer to atomic, qualified, or unqualified structure” or “pointer to atomic, qualified, or unqualified union”, and the second operand shall name a member of the type pointed to.

**Semantics**

3. A postfix expression followed by the `. ` operator and an identifier designates a member of a structure or union object. The value is that of the named member,\(^\text{104}\) and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type

---

\(^\text{103}\)In other words, function executions do not “interleave” with each other.

\(^\text{104}\)If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called “type punning”). This might be a trap representation.
of the designated member.

A postfix expression followed by the \texttt{-}\texttt{> operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.\footnote{If \texttt{&E} is a valid pointer expression (where \texttt{&} is the “address-of” operator, which generates a pointer to its operand), the expression \texttt{(&E)\rightarrow MOS} is the same as \texttt{E.MOS}.} If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

Accessing a member of an atomic structure or union object results in undefined behavior.\footnote{For example, a data race would occur if access to the entire structure or union in one thread conflicts with access to a member from another thread, where at least one access is a modification. Members can be safely accessed using a non-atomic object which is assigned to or from the atomic object.}

One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the completed type of the union is visible. Two structures share a \textit{common initial sequence} if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

\begin{example}
If \texttt{f} is a function returning a structure or union, and \texttt{x} is a member of that structure or union, \texttt{f().x} is a valid postfix expression but is not an lvalue.
\end{example}

\begin{example}
In:

\begin{verbatim}
struct s { int i; const int ci; }; struct s s;
const struct s cs;
volatile struct s vs;
\end{verbatim}

the various members have the types:

\begin{verbatim}
  s.i int  
  s.ci const int  
  cs.i const int  
  cs.ci const int  
  vs.i volatile int  
  vs.ci volatile const int
\end{verbatim}

\begin{example}
The following is a valid fragment:

\begin{verbatim}
union {
  struct {
    int alltypes;
  } n;
  struct {
    int type;
    int intnode;
  } ni;
  struct {
    int type;
    double doublenode;
  } nf;
} u;

u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
  if (sin(u.nf.doublenode) == 0.0)
    /* ... */
\end{verbatim}

The following is not a valid fragment (because the union type is not visible within function \texttt{f}):

\begin{verbatim}
struct t1 { int m; };
struct t2 { int m; };
\end{verbatim}
\end{example}
int f(struct t1 *p1, struct t2 *p2)
{
    if (p1->m < 0)
        p2->m = -p2->m;
    return p1->m;
}
int g()
{
    union {
        struct t1 s1;
        struct t2 s2;
    } u;
    /* ...
    */
    return f(&u.s1, &u.s2);
}

Forward references: address and indirection operators (6.5.3.2 ), structure and union specifiers (6.7.2.1).

6.5.2.4 Postfix increment and decrement operators

Constraints
1 The operand of the postfix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.

Semantics
2 The result of the postfix ++ operator is the value of the operand. As a side effect, the value of the operand object is incremented (that is, the value 1 of the appropriate type is added to it). See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. The value computation of the result is sequenced before the side effect of updating the stored value of the operand. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. Postfix ++ on an object with atomic type is a read-modify-write operation with memory_order_seq_cst memory order semantics.  

3 The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

Forward references: additive operators (6.5.6 ), compound assignment (6.5.16.2 ).

6.5.2.5 Compound literals

Constraints
1 The type name shall specify a complete object type or an array of unknown size, but not a variable length array type.
2 All the constraints for initializer lists in 6.7.9 also apply to compound literals.

\[\text{where a pointer to an atomic object can be formed and } E \text{ has integer type, } E++ \text{ is equivalent to the following code sequence where } T \text{ is the type of } E:\]

\[
T \text{ *addr } = \&E;
T \text{ old } = \text{ *addr};
T \text{ new};
do
\{ new = old + 1;
\} \text{ while } (!\text{ atomic_compare_exchange_strong(addr, \&old, new)});
\]

with old being the result of the operation.
Special care is necessary if E has floating type; see 6.5.16.2.
Semantics

A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a **compound literal**. It provides an unnamed object whose value is given by the initializer list.\(^{108}\)

If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.9, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is a lvalue.

The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.

All the semantic rules for initializer lists in 6.7.9 also apply to compound literals.\(^{109}\)

String literals, and compound literals with const-qualified types, need not designate distinct objects.\(^{110}\)

**EXAMPLE 1** The file scope definition

```c
int *p = (int []){2, 4};
```

initializes p to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

**EXAMPLE 2** In contrast, in

```c
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){*p};
    /*...*/
}
```

p is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by p and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

**EXAMPLE 3** Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```c
drawline((struct point){.x=1, .y=1},
    (struct point){.x=3, .y=4});
```

Or, if `drawline` instead expected pointers to `struct point`:

```c
drawline(&{struct point}{.x=1, .y=1},
    &{struct point}{.x=3, .y=4});
```

**EXAMPLE 4** A read-only compound literal can be specified through constructions like:

```c
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

**EXAMPLE 5** The following three expressions have different meanings:

```c
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

\(^{108}\)Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or `void` only, and the result of a cast expression is not an lvalue.

\(^{109}\)For example, subobjects without explicit initializers are initialized to zero.

\(^{110}\)This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

§ 6.5.2.5 Language
The first always has static storage duration and has type array of char, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

**EXAMPLE 6** Like string literals, const-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
```
might yield 1 if the literals’ storage is shared.

**EXAMPLE 7** Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
struct int_list { int car; struct int_list *cdr; };  
struct int_list endless_zeros = {0, &endless_zeros};  
eval(endless_zeros);  
```

**EXAMPLE 8** Each compound literal creates only a single object in a given scope:

```
struct s { int i; };  
int f (void)  
{  
    struct s *p = 0, *q;  
    int j = 0;  

    again:  
        q = p, p = &((struct s){ j++ });  
        if (j < 2) goto again;  

    return p == q && q->i == 1;  
}
```
The function f() always returns the value 1.

Note that if an iteration statement were used instead of an explicit `goto` and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

Forward references:  type names (6.7.7), initialization (6.7.9).

### 6.5.3 Unary operators

**Syntax**

```
unary-expression:
  postfix-expression
  ++ unary-expression
  -- unary-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-name )
  _Alignof ( type-name )
```

*unary-operator: one of

```
& ++ -- ! & * + - ~
```

### 6.5.3.1 Prefix increment and decrement operators

**Constraints**

The operand of the prefix increment or decrement operator shall have atomic, qualified, or unqualified real or pointer type, and shall be a modifiable lvalue.
Semantics

2 The value of the operand of the prefix `++` operator is incremented. The result is the new value of the operand after incrementation. The expression `++E` is equivalent to `(E+=1)`. See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers.

3 The prefix `--` operator is analogous to the prefix `++` operator, except that the value of the operand is decremented.

Forward references: additive operators (6.5.6), compound assignment (6.5.16.2).

6.5.3.2 Address and indirection operators

Constraints

1 The operand of the unary `&` operator shall be either a function designator, the result of a `[ ]` or unary `*` operator, or an lvalue that designates an object that is not a bit-field and is not declared with the `register` storage-class specifier.

2 The operand of the unary `*` operator shall have pointer type.

Semantics

3 The unary `&` operator yields the address of its operand. If the operand has type “type”, the result has type “pointer to type”. If the operand is the result of a unary `*` operator, neither that operator nor the `&` operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a `[ ]` operator, neither the `&` operator nor the unary `*` that is implied by the `[ ]` is evaluated and the result is as if the `&` operator were removed and the `[ ]` operator were changed to a `+` operator. Otherwise, the result is a pointer to the object or function designated by its operand.

4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type “pointer to type”, the result has type “type”. If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.\(^{111}\)

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

6.5.3.3 Unary arithmetic operators

Constraints

1 The operand of the unary `+` or `-` operator shall have arithmetic type; of the `~` operator, integer type; of the `!` operator, scalar type.

Semantics

2 The result of the unary `+` operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

3 The result of the unary `-` operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.

4 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type `int`. The expression `!E` is

\(^{111}\)Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&{E1[E2]}` to `{(E1)+E2)}. It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `&E` is a function designator or an lvalue equal to `E`. If `P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

§ 6.5.3.3 Language 73
6.5.3.4 The `sizeof` and `_Alignof` operators

**Constraints**

1. The `sizeof` operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. The `_Alignof` operator shall not be applied to a function type or an incomplete type.

**Semantics**

2. The `sizeof` operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

3. The `_Alignof` operator yields the alignment requirement of its operand type. The operand is not evaluated and the result is an integer constant. When applied to an array type, the result is the alignment requirement of the element type.

4. When `sizeof` is applied to an operand that has type `char`, `unsigned char`, or `signed char`, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.112) When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

5. The value of the result of both operators is implementation-defined, and its type (an unsigned integer type) is `size_t`, defined in `<stddef.h>` (and other headers).

6. **EXAMPLE 1** A principal use of the `sizeof` operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to `void`. For example:

```c
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the `alloc` function presumably ensures that its return value is aligned suitably for conversion to a pointer to `double`.

7. **EXAMPLE 2** Another use of the `sizeof` operator is to compute the number of elements in an array:

```c
sizeof array / sizeof array[0]
```

8. **EXAMPLE 3** In this example, the size of a variable length array is computed and returned from a function:

```c
#include <stddef.h>

size_t fsize3(int n)
{
    char b[n+3]; // variable length array
    return sizeof b; // execution time sizeof
}

int main()
{
    size_t size;
    size = fsize3(10); // fsize3 returns 13
    return 0;
}
```

Forward references: common definitions `<stdio.h>` (7.19), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.7), array declarators (6.7.6.2).

112) When applied to a parameter declared to have array or function type, the `sizeof` operator yields the size of the adjusted (pointer) type (see 6.9.1).
### 6.5.4 Cast operators

#### Syntax

1. `cast-expression: unary-expression ( type-name ) cast-expression`

#### Constraints

2. Unless the type name specifies a void type, the type name shall specify atomic, qualified, or unqualified scalar type, and the operand shall have scalar type.

3. Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

4. A pointer type shall not be converted to any floating type. A floating type shall not be converted to any pointer type.

#### Semantics

5. Preceding an expression by a parenthesized type name converts the value of the expression to the unqualified version of the named type. This construction is called a cast. A cast that specifies no conversion has no effect on the type or value of an expression.

6. If the value of the expression is represented with greater range or precision than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type and removes any extra range and precision.

#### Forward references: equality operators (6.5.9), function declarators (including prototypes) (6.7.6.3), simple assignment (6.5.16.1), type names (6.7.7).

### 6.5.5 Multiplicative operators

#### Syntax

1. `multiplicative-expression: cast-expression multiplicative-expression * cast-expression multiplicative-expression / cast-expression multiplicative-expression % cast-expression`

#### Constraints

2. Each of the operands shall have arithmetic type. The operands of the % operator shall have integer type.

3. If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

#### Semantics

4. The usual arithmetic conversions are performed on the operands.

5. The result of the binary * operator is the product of the operands.

6. The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

7. When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded. If the quotient \( a/b \) is representable, the expression \((a/b) \times b + a \% b\) shall equal \(a\);
otherwise, the behavior of both \( a/b \) and \( a\%b \) is undefined.

### 6.5.6 Additive operators

**Syntax**

1. `additive-expression:
   - multiplicative-expression
   - additive-expression + multiplicative-expression
   - additive-expression - multiplicative-expression`

**Constraints**

2. For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to a complete object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

3. For subtraction, one of the following shall hold:
   - both operands have arithmetic type;
   - both operands are pointers to qualified or unqualified versions of compatible complete object types; or
   - the left operand is a pointer to a complete object type and the right operand has integer type.

   (Decrementing is equivalent to subtracting 1.)

4. If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

**Semantics**

5. If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

6. The result of the binary `+` operator is the sum of the operands.

7. The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

8. For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

9. When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression \( P \) points to the \( i \)-th element of an array object, the expressions \((P) + N\) (equivalently, \(N + (P)\)) and \((P) - N\) (where \(N\) has the value \(n\)) point to, respectively, the \(i + n\)-th and \(i - n\)-th elements of the array object, provided they exist. Moreover, if the expression \( P \) points to the last element of an array object, the expression \((P) + 1\) points one past the last element of the array object, and if the expression \( Q \) points one past the last element of an array object, the expression \((Q) - 1\) points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated.

10. When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions \( P \) and \( Q \) point to, respectively, the
\(i\)-th and \(j\)-th elements of an array object, the expression \((P)-(Q)\) has the value \(i - j\) provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression \(P\) points either to an element of an array object or one past the last element of an array object, and the expression \(Q\) points to the last element of the same array object, the expression \(((Q)+1)-(P)\) has the same value as \(((Q)-(P))+1\) and as \(-((P)-((Q)+1))\), and has the value zero if the expression \(P\) points one past the last element of the array object, even though the expression \((Q)+1\) does not point to an element of the array object.\(^\text{115}\)

**EXAMPLE**  Pointer arithmetic is well defined with pointers to variable length array types.

```c
{  
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a;  // p == &a[0]
    p += 1;  // p == &a[1]
    n = p - a;  // n == 1
}
```

If array \(a\) in the above example were declared to be an array of known constant size, and pointer \(p\) were declared to be a pointer to an array of the same known constant size (pointing to \(a\)), the results would be the same.

**Forward references:**  array declarators (6.7.6.2), common definitions `<stddef.h>` (7.19).

### 6.5.7 Bitwise shift operators

**Syntax**

1. `shift-expression:
   
   additive-expression

   `shift-expression`, `shift-expression`, `additive-expression`

   `shift-expression`, `shift-expression`, `additive-expression`

**Constraints**

2. Each of the operands shall have integer type.

**Semantics**

3. The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4. The result of \(E_1 \ll E_2\) is \(E_1\) left-shifted \(E_2\) bit positions; vacated bits are filled with zeros. If \(E_1\) has an unsigned type, the value of the result is \(E_1 \times 2^{E_2}\), reduced modulo one more than the maximum value representable in the result type. If \(E_1\) has a signed type and nonnegative value, and \(E_1 \times 2^{E_2}\) is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5. The result of \(E_1 \gg E_2\) is \(E_1\) right-shifted \(E_2\) bit positions. If \(E_1\) has an unsigned type or if \(E_1\) has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of \(E_1/2^{E_2}\). If \(E_1\) has a signed type and a negative value, the resulting value is implementation-defined.

### 6.5.8 Relational operators

**Syntax**

1. `relational-expression:

   `shift-expression`

   `relational-expression \,< \, shift-expression`

\(^{115}\) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integer expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which can overlap another object in the program) just after the end of the object in order to satisfy the “one past the last element” requirements.
Constraints
2 One of the following shall hold:
   — both operands have real type; or
   — both operands are pointers to qualified or unqualified versions of compatible object types.
3 If either operand has decimal floating type, the other operand shall not have standard floating type.

Semantics
4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.
5 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.
6 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression \( P \) points to an element of an array object and the expression \( Q \) points to the last element of the same array object, the pointer expression \( Q+1 \) compares greater than \( P \). In all other cases, the behavior is undefined.
7 Each of the operators < (less than), > (greater than), \(\leq\) (less than or equal to), and \(\geq\) (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.\(^{116}\) The result has type int.

6.5.9 Equality operators

Syntax
1 equality-expression:
   relational-expression
   equality-expression \(==\) relational-expression
   equality-expression !\(=\) relational-expression

Constraints
2 One of the following shall hold:
   — both operands have arithmetic type;
   — both operands are pointers to qualified or unqualified versions of compatible types;
   — one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of void; or
   — one operand is a pointer and the other is a null pointer constant.
3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

\(^{116}\)The expression \(a<b<c\) is not interpreted as in ordinary mathematics. As the syntax indicates, it means \((a<b)<c\); in other words, “if \(a\) is less than \(b\), compare 1 to \(c\); otherwise, compare 0 to \(c\)”.

Language § 6.5.9
Semantics

4 The == (equal to) and != (not equal to) operators are analogous to the relational operators except for their lower precedence.\(^{117}\) Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type `int`. For any pair of operands, exactly one of the relations is true.

5 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

6 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of `void`, the former is converted to the type of the latter.

7 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.\(^{118}\)

8 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

6.5.10 Bitwise AND operator

Syntax

1 \[\text{AND-expression:} \quad \text{equality-expression} \quad \text{AND-expression} \quad \& \quad \text{equality-expression}\]

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

6.5.11 Bitwise exclusive OR operator

Syntax

1 \[\text{exclusive-OR-expression:} \quad \text{AND-expression} \quad \text{exclusive-OR-expression} \quad \text{AND-expression} \quad \text{AND-expression} \quad \text{AND-expression}\]

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

\(^{117}\)Because of the precedences, `a < b == c < d` is 1 whenever `a < b` and `c < d` have the same truth-value.

\(^{118}\)Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

§ 6.5.11 Language 79
The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

6.5.12 Bitwise inclusive OR operator

Syntax

1  inclusive-OR-expression:
   exclusive-OR-expression
   inclusive-OR-expression | exclusive-OR-expression

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.
4 The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

6.5.13 Logical AND operator

Syntax

1  logical-AND-expression:
   inclusive-OR-expression
   logical-AND-expression && logical-AND-expression

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The && operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type int.
4 Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

6.5.14 Logical OR operator

Syntax

1  logical-OR-expression:
   logical-AND-expression
   logical-AND-expression || logical-AND-expression

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The || operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type int.
4 Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.
6.5.15 Conditional operator

Syntax

- \texttt{conditional-expression:}
  - \texttt{logical-OR-expression}
  - \texttt{logical-OR-expression ? expression : conditional-expression}

Constraints

1. The first operand shall have scalar type.
2. One of the following shall hold for the second and third operands:
   - both operands have arithmetic type;
   - both operands have the same structure or union type;
   - both operands have void type;
   - both operands are pointers to qualified or unqualified versions of compatible types;
   - one operand is a pointer and the other is a null pointer constant; or
   - one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of \texttt{void}.
3. If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics

4. The first operand is evaluated; there is a sequence point between its evaluation and the evaluation of the second or third operand (whichever is evaluated). The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.\footnote{A conditional expression does not yield an lvalue.}
5. If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.
6. If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types referenced by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to \texttt{void} or a qualified version of \texttt{void}, in which case the result type is a pointer to an appropriately qualified version of \texttt{void}.
7. \texttt{EXAMPLE} The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.
8. \texttt{EXAMPLE} The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types.

\begin{sourcecode}

\begin{verbatim}
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
\end{verbatim}

\end{sourcecode}
the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

<table>
<thead>
<tr>
<th>c vp</th>
<th>c ip</th>
<th>const void *</th>
</tr>
</thead>
<tbody>
<tr>
<td>v ip</td>
<td>0</td>
<td>volatile int *</td>
</tr>
<tr>
<td>c ip</td>
<td>v ip</td>
<td>const volatile int *</td>
</tr>
<tr>
<td>vp</td>
<td>c cp</td>
<td>const void *</td>
</tr>
<tr>
<td>ip</td>
<td>c ip</td>
<td>const int *</td>
</tr>
<tr>
<td>vp</td>
<td>ip</td>
<td>void *</td>
</tr>
</tbody>
</table>

### 6.5.16 Assignment operators

#### Syntax

```
assignment-expression:
  conditional-expression
  unary-expression assignment-operator assignment-expression

assignment-operator:
  one of
    =  *= /= %= += -= <<= >>= &= ^= |=
```

#### Constraints

1. An assignment operator shall have a modifiable lvalue as its left operand.

#### Semantics

2. An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type the left operand would have after lvalue conversion. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

### 6.5.16.1 Simple assignment

#### Constraints

1. One of the following shall hold:

   - the left operand has atomic, qualified, or unqualified arithmetic type, and the right has arithmetic type;
   - the left operand has an atomic, qualified, or unqualified version of a structure or union type compatible with the type of the right;
   - the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;
   - the left operand has atomic, qualified, or unqualified pointer type, and (considering the type the left operand would have after lvalue conversion) one operand is a pointer to an object type, and the other is a pointer to a qualified or unqualified version of `void`, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;
   - the left operand is an atomic, qualified, or unqualified `_Bool`, and the right is a pointer.

---

The implementation is permitted to read the object to determine the value but is not required to, even when the object has volatile-qualified type.

The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to “the value of the expression” and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes `const` but not `volatile` from the type `int volatile * const`).
Semantics
2 In simple assignment (=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.
3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

EXAMPLE 1 In the program fragment

```c
int f(void);
char c;
/* ... */
if ((c = f()) == -1)
    /* ... */
```

the int value returned by the function could be truncated when stored in the char, and then converted back to int width prior to the comparison. In an implementation in which “plain” char has the same range of values as unsigned char (and char is narrower than int), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable c would be declared as int.

EXAMPLE 2 In the fragment:

```c
char c;
int i;
long l;
l = (c = i);
```

the value of i is converted to the type of the assignment expression c = i, that is, char type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, long int type.

EXAMPLE 3 Consider the fragment:

```c
const char **cpp;
char *p;
const char c = 'A';
cpp = &p; // constraint violation
*p = 65;   // valid
*p = 0;    // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object c.

6.5.16.2 Compound assignment

Constraints
1 For the operators += and -= only, either the left operand shall be an atomic, qualified, or unqualified pointer to a complete object type, and the right shall have integer type; or the left operand shall have atomic, qualified, or unqualified arithmetic type, and the right shall have arithmetic type.
2 For the other operators, the left operand shall have atomic, qualified, or unqualified arithmetic type, and (considering the type the left operand would have after lvalue conversion) each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.
3 If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

Semantics
4 A compound assignment of the form E1 op= E2 is equivalent to the simple assignment expression E1 = E1 (E2), except that the lvalue E1 is evaluated only once, and with respect to an inde-terminate-sequenced function call, the operation of a compound assignment is a single evaluation. If E1 has an atomic type, compound assignment is a read-modify-write operation with memory_order_seq_cst memory order semantics.
NOTE  Where a pointer to an atomic object can be formed and \( E_1 \) and \( E_2 \) have integer type, this is equivalent to the following code sequence where \( T_1 \) is the type of \( E_1 \) and \( T_2 \) is the type of \( E_2 \):

\[
\begin{align*}
T_1\ \star \text{addr} &= &\&E_1; \\
T_2\ \text{val} &= (E_2); \\
T_1\ \text{old} &= \star\text{addr}; \\
T_1\ \text{new}; \\
\text{do} \{ \\
\quad \text{new} &= \text{old} \text{ op} \text{ val}; \\
\} \text{ while} \ (\text{!atomic_compare_exchange_strong}\ (\text{addr}, &\text{old}, \text{new});
\end{align*}
\]

with \( \text{new} \) being the result of the operation.

If \( E_1 \) or \( E_2 \) has floating type, then exceptional conditions or floating-point exceptions encountered during discarded evaluations of \( \text{new} \) would also be discarded in order to satisfy the equivalence of \( E_1 \text{ op } E_2 \) and \( E_1 = E_1 \text{ op } (E_2) \). For example, if Annex F is in effect, the floating types involved have IEC 60559 formats, and \texttt{FLT_EVAL_METHOD} is 0, the equivalent code would be:

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
/* ...
fenvt fenv;
T1 \star addr = &E1;
T2 \text{val} = E2;
T1 \text{old} = \star\text{addr};
T1 \text{new};
feholdexcept(&fenv);
for (; ;) { \\
\quad \text{new} = \text{old} \text{ op } \text{val}; \\
\quad \text{if} (\text{atomic}\_\text{compare}\_\text{exchange}\_\text{strong}(\text{addr}, &\text{old}, \text{new})) \\
\quad \quad \text{break}; \\
\quad \text{feclearexcept}(\text{FE\_ALL\_EXCEPT}); \\
\}
feupdateenv(&fenv);
```

If \texttt{FLT\_EVAL\_METHOD} is not 0, then \( T_2 \) is expected to be a type with the range and precision to which \( E_2 \) is evaluated in order to satisfy the equivalence.

### 6.5.17 Comma operator

**Syntax**

1. `expression`: `assignment-expression expression , assignment-expression`

**Semantics**

2. The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value.\(^{122}\)

3. **EXAMPLE** As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

\[
f(a, (t=3, t+2), c)
\]

the function has three arguments, the second of which has the value 5.

**Forward references**: initialization (6.7.9).

\(^{122}\)A comma operator does not yield an lvalue.
6.6 Constant expressions

Syntax

constant-expression:
  conditional-expression

Description

A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

- Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.\(^\text{123}\)
- Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Semantics

- An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic range and precision shall be at least as great as if the expression were being evaluated in the execution environment.\(^\text{124}\)
- An integer constant expression\(^\text{125}\) shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, _Alignof expressions, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the sizeof or _Alignof operator.

More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:

  - an arithmetic constant expression,
  - a null pointer constant,
  - an address constant, or
  - an address constant for a complete object type plus or minus an integer constant expression.

- An arithmetic constant expression shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, sizeof expressions whose results are integer constants, and _Alignof expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a sizeof or _Alignof operator.

- An address constant is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript \([\ ]\) and member-access \(\rightarrow\) operators, the address & and indirection \(^*\) unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

An implementation may accept other forms of constant expressions.

\(^{123}\)The operand of a sizeof or _Alignof operator is usually not evaluated (6.5.3.4).

\(^{124}\)The use of evaluation formats as characterized by FLT_EVAL_METHOD also applies to evaluation in the translation environment.

\(^{125}\)An integer constant expression is required in a number of contexts such as the size of a bit-field member of a structure, the value of an enumeration constant, and the size of a non-variable length array. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.
The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.\footnote{Thus, in the following initialization,
\begin{verbatim}
static int i = 2 || 1 / 0;
\end{verbatim}
the expression is a valid integer constant expression with value one.}

\textbf{Forward references:} array declarators (6.7.6.2), initialization (6.7.9).
6.7 Declarations

Syntax
1 declaration:
   declaration-specifiers init-declarator-list opt ;
   attribute-specifier-sequence declaration-specifiers init-declarator-list ;
   static_assert-declaration

declaration-specifiers:
   storage-class-specifier declaration-specifiers declaration-specifier attribute-specifier-sequence opt
   type-specifier declaration-specifiers declaration-specifier declaration-specifiers

declaration-specifier:
   type-qualifier declaration-specifiers opt
   function-specifier declaration-specifiers type-specifier-qualifier
   alignment-specifier declaration-specifiers opt
   function-specifier

init-declarator-list:
   init-declarator
   init-declarator-list , init-declarator

init-declarator:
   declarator
   declarator = initializer

attribute-declaration:
   attribute-specifier-sequence ;

Constraints
2 A declaration other than a static_assert or attribute declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:
   — a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;
   — tags may be redeclared as specified in 6.7.2.3.

4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics
5 A declaration specifies the interpretation and attributes of a set of identifiers. A definition of an identifier is a declaration for that identifier that:
   — for an object, causes storage to be reserved for that object;
   — for a function, includes the function body;\textsuperscript{127}
   — for an enumeration constant, is the (only) declaration of the identifier;
   — for a typedef name, is the first (or only) declaration of the identifier.

6 The declaration specifiers consist of a sequence of specifiers, followed by an optional attribute specifier sequence, that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator list init declarator list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared. The optional attribute specifier sequence appertains to each of the entities declared by the declarators of the init declarator list.

\textsuperscript{127}Function definitions have a different syntax, described in 6.9.1.
If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters (including in prototypes), it is the adjusted type (see 6.7.6.3) that is required to be complete.

The optional attribute specifier sequence terminating a sequence of declaration specifiers appertains to the type determined by the preceding sequence of declaration specifiers. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

Except where specified otherwise, the meaning of an attribute declaration is implementation-defined.

EXAMPLE In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the identifier for that declaration.

```c
[[deprecated]] void f [[deprecated]] (void); // valid
```

Forward references: declarsers (6.7.6), enumeration specifiers (6.7.2.2), initialization (6.7.9), type names (6.7.7), type qualifiers (6.7.3).

### 6.7.1 Storage-class specifiers

#### Syntax

```c
storage-class-specifier:
    typedef
    extern
    static
    __Thread_local
    auto
    register
```

#### Constraints

At most, one storage-class specifier may be given in the declaration specifiers in a declaration, except that `__Thread_local` may appear with `static` or `extern`.\(^\text{128}\)

In the declaration of an object with block scope, if the declaration specifiers include `__Thread_local`, they shall also include either `static` or `extern`. If `__Thread_local` appears in any declaration of an object, it shall be present in every declaration of that object.

`__Thread_local` shall not appear in the declaration specifiers of a function declaration.

#### Semantics

The `typedef` specifier is called a “storage-class specifier” for syntactic convenience only; it is discussed in 6.7.8. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.

A declaration of an identifier for an object with storage-class specifier `register` suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.\(^\text{129}\)

The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than `extern`.

If an aggregate or union object is declared with a storage-class specifier other than `typedef`, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the

---

\(^{128}\)See “future language directions” (6.11.5).

\(^{129}\)The implementation can treat any `register` declaration simply as an `auto` declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier `register` cannot be computed, either explicitly (by use of the unary `&` operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier `register` is `sizeof`.
members of the object, and so on recursively for any aggregate or union member objects.

Forward references: type definitions (6.7.8).

6.7.2 Type specifiers

Syntax

1

type-specifier:

  void
  char
  short
  int
  long
  float
  double
  signed
  unsigned
  _Bool
  _Complex
  _Decimal32
  _Decimal64
  _Decimal128
  atomic-type-specifier
  struct-or-union-specifier
  enum-specifier
  typedef-name

Constraints

2

At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct-member declaration and type name. Each list of type specifiers shall be one of the following multisets (delimited by commas, when there is more than one multiset per item); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

— void
— char
— signed char
— unsigned char
— short, signed short, short int, or signed short int
— unsigned short, or unsigned short int
— int, signed, or signed int
— unsigned, or unsigned int
— long, signed long, long int, or signed long int
— unsigned long, or unsigned long int
— long long, signed long long, long long int, or signed long long int
— unsigned long long, or unsigned long long int
— float
— double
— long double
— _Decimal32
— _Decimal64
The type specifier `__Complex` shall not be used if the implementation does not support complex types, and the type specifiers `__Decimal32`, `__Decimal64`, and `__Decimal128` shall not be used if the implementation does not support decimal floating types (see 6.10.8.3).

Semantics

Specifiers for structures, unions, enumerations, and atomic types are discussed in 6.7.2.1 through 6.7.2.4, 6.7.2.1 through 6.7.2.4. Declarations of typedef names are discussed in 6.7.8. The characteristics of the other types are discussed in 6.2.5.

Each of the comma-separated multisets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier `int` designates the same type as `signed int` or the same type as `unsigned int`.

Forward references: atomic type specifiers (6.7.2.4), enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.8).

6.7.2.1 Structure and union specifiers

Syntax

```c
struct-or-union-specifier:
  struct-or-union identifier opt { struct-declaration-list } struct-or-union_attribute-specifier-sequence opt identifier

struct-or-union:
  struct
  union

struct-declaration-list:
  struct-declaration member-declaration-list
  struct-declaration-list struct-declaration member-declaration-list

struct-declaration:
  specifier-qualifier-list member-declaration-list opt ;
  static_assert-declaration

specifier-qualifier-list:
  type-specifier specifier-qualifier-list type-specifier-qualifier_attribute-specifier-sequence opt
  type-qualifier specifier-qualifier-list type-specifier-qualifierSpecifier-qualifier-list

type-specifier-qualifier:
  typespecifier
  type-qualifier
```
a structure or union shall not contain a member with incomplete or function type (hence, a structure
Structure and union specifiers have the same form. The keywords
A bit-field shall have a type that is a qualified or unqualified version of _Bool, signed int, unsigned int, or some other implementation-defined type. It is implementation-defined whether atomic types are permitted.
An attribute specifier sequence shall not appear in a struct-or-union specifier without a member
declarator list, except in a declaration of the form:

The attributes in the attribute specifier sequence, if any, are thereafter considered attributes of the
struct or union whenever it is named.

As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is
allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose
storage overlap.
Structure and union specifiers have the same form. The keywords struct and union indicate that
the type being specified is, respectively, a structure type or a union type.
The optional attribute specifier sequence in a struct-or-union specifier appertains to the structure
or union type being declared. The optional attribute specifier sequence in a member declaration
appertains to each of the members declared by the member declarator list; it shall not appear if the
optional member declarator list is omitted. The optional attribute specifier sequence in a specifier
qualifier list appertains to the type denoted by the preceding type specifier qualifiers. The attribute
specifier sequence affects the type only for the member declaration or type name it appears in, not
other types or declarations involving the same type.

The presence of a struct-declaration list in a struct-or-union specifier member declaration list in a
struct-or-union specifier declares a new type, within a translation unit. The struct-declaration list
member declaration list is a sequence of declarations for the members of the structure or union. If the
struct-declaration list-member declaration list does not contain any named members, either

---

130] While the number of bits in a _Bool object is at least CHAR_BIT, the width of a _Bool can be just 1 bit.
directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is incomplete until immediately after the } that terminates the list, and complete thereafter.

A member of a structure or union may have any complete object type other than a variably modified type.\footnote{A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.} In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field\footnote{The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.}; its width is preceded by a colon.

A bit-field is interpreted as having a signed or unsigned integer type consisting of the specified number of bits.\footnote{As specified in 6.7.2 above, if the actual type specifier used is int or a typedef-name defined as int, then it is implementation-defined whether the bit-field is signed or unsigned.} If the value 0 or 1 is stored into a nonzero-width bit-field of type \_Bool, the value of the bit-field shall compare equal to the value stored; a \_Bool bit-field has the semantics of a \_Bool.

An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.\footnote{An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.} As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

An unnamed member whose type specifier is a structure specifier with no tag is called an anonymous structure; an unnamed member whose type specifier is a union specifier with no tag is called an anonymous union. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

There may be unnamed padding at the end of a structure or union.

As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a flexible array member. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a . (or -> ) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.
EXAMPLE 1 The following declarations illustrate the behavior when an attribute is written on a tag declaration:

```c
struct [[deprecated]] S; // valid, [[deprecated]] appertains to struct S
void f(struct S *s); // valid, the struct S type has the [[deprecated]] attribute  
                      // from the previous declaration
struct S {  
    int a;  // from the previous declaration
};
void g(struct [[deprecated]] S s); // invalid
```

EXAMPLE 2 The following illustrates anonymous structures and unions:

```c
struct v {  
    union {  
        struct { int i, j; }; // anonymous structure  
        struct { long k, l; } w;
    }
    int m;
} v1;

v1.i = 2; // valid
v1.k = 3; // invalid: inner structure is not anonymous
v1.w.k = 5; // valid
```

EXAMPLE 3 After the declaration:

```c
struct s { int n; double d[]; }
```

the structure `struct s` has a flexible array member `d`. A typical way to use this is:

```c
int m = /* some value */;
struct s *p = malloc(sizeof(struct s) + sizeof(double [m]));
```

and assuming that the call to `malloc` succeeds, the object pointed to by `p` behaves, for most purposes, as if `p` had been declared as:

```c
struct { int n; double d[m]; } *p;
```

(there are circumstances in which this equivalence is broken; in particular, the offsets of member `d` might not be the same).

Following the above declaration:

```c
struct s t1 = { 0 }; // valid
struct s t2 = { 1, { 4.2 } }; // invalid
  t1.n = 4;  // valid
  t1.d[0] = 4.2; // might be undefined behavior
```

The initialization of `t2` is invalid (and violates a constraint) because `struct s` is treated as if it did not contain member `d`. The assignment to `t1.d[0]` is probably undefined behavior, but it is possible that

```c
sizeof (struct s) >= offsetof(struct s, d) + sizeof (double)
```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

After the further declaration:

```c
struct ss { int n; }
```

the expressions:

```c
sizeof (struct s) >= sizeof (struct ss)  
sizeof (struct s) >= offsetof(struct s, d)
```

are always equal to 1.

If `sizeof (double)` is 8, then after the following code is executed:

```c
```
struct *s1;
struct *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);

and assuming that the calls to malloc succeed, the objects pointed to by s1 and s2 behave, for most purposes, as if the identifiers had been declared as:

```c
struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;
```

Following the further successful assignments:

```c
s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) + 6);
```

they then behave as if the declarations were:

```c
struct { int n; double d[1]; } *s1, *s2;
```

and:

```c
double *dp;
dp = &s1->d[0]; // valid
*dp = 42;       // valid
dp = &s2->d[0]; // valid
*dp = 42;       // undefined behavior
```

The assignment:

```c
*s1 = *s2;
```

only copies the member n; if any of the array elements are within the first sizeof (struct s) bytes of the structure, they might be copied or simply overwritten with indeterminate values.

**EXAMPLE 4** Because members of anonymous structures and unions are considered to be members of the containing structure or union, struct s in the following example has more than one named member and thus the use of a flexible array member is valid:

```c
struct s {
    struct { int i; };
    int a[];
};
```

**Forward references:** declarators (6.7.6), tags (6.7.2.3).

### 6.7.2.2 Enumeration specifiers

#### Syntax

1. `enum-specifier:`

   ```c
   enum attribute-specifier-sequence\_opt identifier\_opt { enumerator-list } 
   enum attribute-specifier-sequence\_opt identifier\_opt { enumerator-list , } 
   enum identifier
   ```

2. `enumerator-list:`

   ```c
   enumerator 
   enumerator-list , enumerator
   ```

3. `enumerator:`

   ```c
   enumeration-constant attribute-specifier-sequence\_opt 
   enumeration-constant attribute-specifier-sequence\_opt = constant-expression
   ```
Constraints

1. The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an `int`.

Semantics

2. The optional attribute specifier sequence in the `enum` specifier appertains to the enumeration; the attributes in that attribute specifier sequence are thereafter considered attributes of the enumeration whenever it is named. The optional attribute specifier sequence in the enumerator appertains to that enumerator.

3. The identifiers in an enumerator list are declared as constants that have type `int` and may appear wherever such are permitted. An enumerator with `=` defines its enumeration constant as the value of the constant expression. If the first enumerator has no `=`, the value of its enumeration constant is 0. Each subsequent enumerator with no `=` defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with `=` may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

4. Each enumerated type shall be compatible with `char`, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined, but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until immediately after the `}` that terminates the list of enumerator declarations, and complete thereafter.

EXAMPLE

```c
enum hue { chartreuse, burgundy, claret=20, winedark };
enum hue col, *cp;
    col = claret;
    cp = &col;
if (*cp != burgundy)
    /* ... */
```

makes `hue` the tag of an enumeration, and then declares `col` as an object that has that type and `cp` as a pointer to an object that has that type. The enumerated values are in the set `{0, 1, 20, 21}`.

Forward references: tags (6.7.2.3).

6.7.2.3 Tags

Constraints

1. A specific type shall have its content defined at most once.

2. Where two declarations that use the same tag declare the same type, they shall both use the same choice of `struct`, `union`, or `enum`.

3. A type specifier of the form

   ```c
   enum identifier
   ```

   without an enumerator list shall only appear after the type it specifies is complete.

4. A type specifier of the form

   ```c
   struct-or-union attribute-specifier-sequence opt identifier
   ```

   shall not contain an attribute specifier sequence.\(^{137}\)

Semantics

5. All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. Irrespective of whether there is a tag or what other declarations of the

\(^{135}\) Thus, the identifiers of enumeration constants declared in the same scope are all required to be distinct from each other and from other identifiers declared in ordinary declarators.

\(^{136}\) An implementation can delay the choice of which integer type until all enumeration constants have been seen.

\(^{137}\) As specified in 6.7.2.1 above, the type specifier may be followed by a `;` or a member declaration list.
Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.

A type specifier of the form

```
struct-or-union _attribute-specifier-sequence_opt identifier_opt { _struct-declaration-list_ }
```

or

```
enum _attribute-specifier-sequence_opt identifier_opt { enumerator-list }
```

declarates a structure, union, or enumerated type. The list defines the structure content, union content, or enumeration content. If an identifier is provided, the type specifier also declares the identifier to be the tag of that type. The optional attribute specifier sequence appertains to the structure, union, or enumeration type being declared; the attributes in that attribute specifier sequence are thereafter considered attributes of the structure or union, or enumeration type whenever it is named.

A declaration of the form

```
struct-or-union _attribute-specifier-sequence_opt identifier ;
```

specifies a structure or union type and declares the identifier as a tag of that type. The optional attribute specifier sequence appertains to the structure or union type being declared; the attributes in that attribute specifier sequence are there after considered attributes of the structure or union type whenever it is named.

If a type specifier of the form

```
struct-or-union _attribute-specifier-sequence_opt identifier
```

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.

If a type specifier of the form

```
struct-or-union _attribute-specifier-sequence_opt identifier
```

or

```
enum identifier
```

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redefine the tag.

**EXAMPLE 1** This mechanism allows declaration of a self-referential structure.

```
struct tnode {
    int count;
    struct tnode *left, *right;
};
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been

---

138) An incomplete type can only be used when the size of an object of that type is not needed. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.) The specification has to be complete before such a function is called or defined.

139) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

140) A similar construction with enum does not exist.
given, the declaration

```c
struct tnode s, *sp;
```

declares `s` to be an object of the given type and `sp` to be a pointer to an object of the given type. With these declarations, the expression `sp->left` refers to the left `struct tnode` pointer of the object to which `sp` points; the expression `s.right->count` designates the `count` member of the right `struct tnode` pointed to from `s`.

The following alternative formulation uses the `typedef` mechanism:

```c
typedef struct tnode TNODE;
struct tnode {
    int count;
    TNODE *left, *right;
};
TNODE s, *sp;
```

**EXAMPLE 2** To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

```c
struct s1 { struct s2 *s2p; /* ... */ }; // D1
struct s2 { struct s1 *s1p; /* ... */ }; // D2
```

specify a pair of structures that contain pointers to each other. Note, however, that if `s2` were already declared as a tag in an enclosing scope, the declaration `D1` would refer to it, not to the tag `s2` declared in `D2`. To eliminate this context sensitivity, the declaration

```c
struct s2;
```

can be inserted ahead of `D1`. This declares a new tag `s2` in the inner scope; the declaration `D2` then completes the specification of the new type.

**Forward references**: declarators (6.7.6), type definitions (6.7.8).

### 6.7.2.4 Atomic type specifiers

**Syntax**

1. `atomic-type-specifier:
   _Atomic (type-name)`

**Constraints**

2. Atomic type specifiers shall not be used if the implementation does not support atomic types (see 6.10.8.3).

3. The type name in an atomic type specifier shall not refer to an array type, a function type, an atomic type, or a qualified type.

**Semantics**

4. The properties associated with atomic types are meaningful only for expressions that are lvalues. If the `_Atomic` keyword is immediately followed by a left parenthesis, it is interpreted as a type specifier (with a type name), not as a type qualifier.

### 6.7.3 Type qualifiers

**Syntax**

1. `type-qualifier:
   const
   restrict
   volatile
   _Atomic`
Constraints

2 Types other than pointer types whose referenced type is an object type shall not be restrict-qualified.

3 The _Atomic qualifier shall not be used if the implementation does not support atomic types (see 6.10.8.3).

4 The type modified by the _Atomic qualifier shall not be an array type or a function type.

Semantics

5 The properties associated with qualified types are meaningful only for expressions that are lvalues.

6 If the same qualifier appears more than once in the same specifier-qualifier list or as declaration specifiers, either directly or via one or more typedefs, the behavior is the same as if it appeared only once. If other qualifiers appear along with the _Atomic qualifier the resulting type is the so-qualified atomic type.

7 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

8 An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously. What constitutes an access to an object that has volatile-qualified type is implementation-defined.

9 An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer. The intended use of the restrict qualifier (like the register storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).

10 If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.

11 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

12 EXAMPLE 1 An object declared

   extern const volatile int real_time_clock;

might be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

13 EXAMPLE 2 The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

   const struct s { int mem; } cs = { 1 };
   struct s ncs; // the object ncs is modifiable

---

141) The implementation can place a const object that is not volatile in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

142) This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

143) A volatile declaration can be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared are not allowed to be “optimized out” by an implementation or reordered except as permitted by the rules for evaluating expressions.

144) For example, a statement that assigns a value returned by malloc to a single pointer establishes this association between the allocated object and the pointer.

145) Both of these can occur through the use of typedefs.
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; // array of array of const int
int *pi;
const int *pci;

ncs = cs;  // valid
cs = ncs;  // violates modifiable lvalue constraint for =
pi = &ncs.mem;  // valid
pi = &cs.mem;  // violates type constraints for =
pci = &cs.mem;  // valid
pi = a[0];  // invalid: a[0] has type "const int *"

**EXAMPLE 3** The declaration

```
_Atomic volatile int *p;
```

specifies that p has the type “pointer to volatile atomic int”, a pointer to a volatile-qualified atomic type.
6.7.3.1 Formal definition of restrict

1 Let $D$ be a declaration of an ordinary identifier that provides a means of designating an object $P$ as a restrict-qualified pointer to type $T$.

2 If $D$ appears inside a block and does not have storage class extern, let $B$ denote the block. If $D$ appears in the list of parameter declarations of a function definition, let $B$ denote the associated block. Otherwise, let $B$ denote the block of main (or the block of whatever function is called at program startup in a freestanding environment).

3 In what follows, a pointer expression $E$ is said to be based on object $P$ if (at some sequence point in the execution of $B$ prior to the evaluation of $E$) modifying $P$ to point to a copy of the array object into which it formerly pointed would change the value of $E$.\(^{(146)}\) Note that “based” is defined only for expressions with pointer types.

4 During each execution of $B$, let $L$ be any lvalue that has &$L$ based on $P$. If $L$ is used to access the value of the object $X$ that it designates, and $X$ is also modified (by any means), then the following requirements apply: $T$ shall not be const-qualified. Every other lvalue used to access the value of $X$ shall also have its address based on $P$. Every access that modifies $X$ shall be considered also to modify $P$, for the purposes of this subclause. If $P$ is assigned the value of a pointer expression $E$ that is based on another restricted pointer object $P_2$, associated with block $B_2$, then either the execution of $B_2$ shall begin before the execution of $B$, or the execution of $B_2$ shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.

5 Here an execution of $B$ means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration associated with $B$.

6 A translator is free to ignore any or all aliasing implications of uses of restrict.

EXAMPLE 1 The file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of $a$, $b$, or $c$, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

EXAMPLE 2 The function parameter declarations in the following example

```c
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0)
        *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other. The translator can make this no-aliasing inference based on the parameter declarations alone, without analyzing the function body.

9 The benefit of the restrict qualifiers is that they enable a translator to make an effective dependence analysis of function $f$ without examining any of the calls of $f$ in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of $f$ in $g$ has undefined behavior because each of $d[1]$ through $d[49]$ is accessed through both $p$ and $q$.

```c
void g(void)
{
    extern int d[100];
    f(50, d + 50, d); // valid
    f(50, d + 1, d); // undefined behavior
}
```

\(^{(146)}\)In other words, $E$ depends on the value of $P$ itself rather than on the value of an object referenced indirectly through $P$. For example, if identifier $p$ has type (int * restrict), then the pointer expressions $p$ and $p+1$ are based on the restricted pointer object designated by $p$, but the pointer expressions $*p$ and $p[1]$ are not.
EXAMPLE 3  The function parameter declarations

```c
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if \(a\) and \(b\) are disjoint arrays, a call of the form \(h(100, a, b, b)\) has defined behavior, because array \(b\) is not modified within function \(h\).

EXAMPLE 4  The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only “outer-to-inner” assignments between restricted pointers declared in nested blocks have defined behavior.

```c
{
    int * restrict p1;
    int * restrict q1;
    p1 = q1; // undefined behavior
    {
        int * restrict p2 = p1; // valid
        int * restrict q2 = q1; // valid
        p1 = q2; // undefined behavior
        p2 = q2; // undefined behavior
    }
}
```

The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`.

```c
typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
    vector t;
    t.n = n;
    t.v = malloc(n * sizeof (float));
    return t;
}
```

EXAMPLE 5  Suppose that a programmer knows that references of the form \(p[i]\) and \(q[i]\) are never aliases in the body of a function:

```c
void f(int n, int*p, int*q) { /* ... */ }
```

There are several ways that this information could be conveyed to a translator using the `restrict` qualifier. Example 2 shows the most effective way, qualifying all pointer parameters, and can be used provided that neither \(p\) nor \(q\) becomes based on the other in the function body. A potentially effective alternative is:

```c
void f(int n, int * restrict p, int * const q) { /* ... */ }
```

Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use subtler reasoning: that the `const`-qualification of \(q\) precludes it becoming based on \(p\). There is also a requirement that \(q\) is not modified, so this alternative cannot be used for the function in Example 2, as written.

EXAMPLE 6  Another potentially effective alternative is:

```c
void f(int n, int *p, int const * restrict q) { /* ... */ }
```

Again it is possible for a translator to make the no-aliasing inference based on the parameter declarations alone, though now it must use even subtler reasoning: that this combination of `restrict` and `const` means that objects referenced using \(q\) cannot be modified, and so no modified object can be referenced using both \(p\) and \(q\).
A function specifier may appear more than once; the behavior is the same as if it appeared only once.

Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an inline function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the inline function specifier without extern, then the definition in that translation unit is an inline definition. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.

A function declared with a _Noreturn function specifier shall not return to its caller.

Recommended practice

The implementation should produce a diagnostic message for a function declared with a _Noreturn function specifier that appears to be capable of returning to its caller.

EXAMPLE 1 The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with extern creates an external definition. The following example shows an entire translation unit.

```c
void f(int n, int * restrict p, int *q) { /* ... */ }
```

Here the translator can make the no-aliasing inference only by analyzing the body of the function and proving that q cannot become based on p. Some translator designs may choose to exclude this analysis, given availability of the more effective alternatives above. Such a translator is required to assume that aliases are present because assuming that aliases are not present may result in an incorrect translation. Also, a translator that attempts the analysis may not succeed in all cases and thus need to conservatively assume that aliases are present.

### 6.7.4 Function specifiers

#### Syntax

```c
function-specifier:
  inline
  _Noreturn
```

#### Constraints

1. Function specifiers shall be used only in the declaration of an identifier for a function.
2. An inline function or definition is distinct from the corresponding external definition and from any other corresponding inline definition(s) in other translation units, all corresponding objects with static storage duration are also distinct in each of the translation units.
3. In a hosted environment, no function specifier(s) shall appear in a declaration of `main`.

#### Semantics

1. A function specifier may appear more than once; the behavior is the same as if it appeared only once.
2. A function declared with an `inline` function specifier is an inline function. Making a function an inline function suggests that calls to the function be as fast as possible. The extent to which such suggestions are effective is implementation-defined.
3. Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an `inline` function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the `inline` function specifier without `extern`, then the definition in that translation unit is an inline definition. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.
4. A function declared with a _Noreturn function specifier shall not return to its caller.

---

147) By using, for example, an alternative to the usual function call mechanism, such as “inline substitution”. Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.
148) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an `inline` declaration.
149) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.
inline double fahr(double t)
{
    return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
    return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double); // creates an external definition

double convert(int is_fahr, double temp)
{
    /* A translator may perform inline substitutions */
    return is_fahr ? cels(temp): fahr(temp);
}

11 Note that the definition of fahr is an external definition because fahr is also declared with extern, but the definition of cels is an inline definition. Because cels has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either can be used for the call.

12 EXAMPLE 2

_Noreturn void f () {
    abort(); // ok
}

_Noreturn void g (int i) { // causes undefined behavior if i <= 0
    if (i > 0) abort();
}

Forward references: function definitions (6.9.1).

6.7.5 Alignment specifier

Syntax

1

alignment-specifier:

_ALIGNas (type-name )
_ALIGNas (constant-expression )

Constraints

2 An alignment specifier shall appear only in the declaration specifiers of a declaration, or in the specifier-qualifier list of a member declaration, or in the type name of a compound literal. An alignment specifier shall not be used in conjunction with either of the storage-class specifiers typedef or register, nor in a declaration of a function or bit-field.

3 The constant expression shall be an integer constant expression. It shall evaluate to a valid fundamental alignment, or to a valid extended alignment supported by the implementation for an object of the storage duration (if any) being declared, or to zero.

4 An object shall not be declared with an over-aligned type with an extended alignment requirement not supported by the implementation for an object of that storage duration.

5 The combined effect of all alignment specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the type of the object or member being declared.

§ 6.7.5 Language 103
Semantics

6 The first form is equivalent to \texttt{\_Alignas(\_Alignof( type-name ))}.

7 The alignment requirement of the declared object or member is taken to be the specified alignment. An alignment specification of zero has no effect.\footnote{An alignment specification of zero also does not affect other alignment specifications in the same declaration.} When multiple alignment specifiers occur in a declaration, the effective alignment requirement is the strictest specified alignment.

8 If the definition of an object has an alignment specifier, any other declaration of that object shall either specify equivalent alignment or have no alignment specifier. If the definition of an object does not have an alignment specifier, any other declaration of that object shall also have no alignment specifier. If declarations of an object in different translation units have different alignment specifiers, the behavior is undefined.

6.7.6 Declarators

Syntax

1 \begin{verbatim}
declarator: pointer_opt direct-declarator

direct-declarator:
  identifier attribute-specifier-sequence_opt
  ( declarator )
  array-declarator attribute-specifier-sequence_opt
  function-declarator attribute-specifier-sequence_opt

array-declarator:
  direct-declarator [ type-qualifier-list_opt assignment-expression_opt ]
  direct-declarator [ static type-qualifier-list_opt assignment-expression ]
  direct-declarator [ type-qualifier-list static assignment-expression ]
  direct-declarator [ type-qualifier-list_opt * ]
  direct-declarator ( parameter-type-list )

function-declarator:
  direct-declarator ( identifier-list_opt parameter-type-list_opt )

pointer:
  * attribute-specifier-sequence_opt type-qualifier-list_opt
  * attribute-specifier-sequence_opt type-qualifier-list_opt pointer

type-qualifier-list:
  type-qualifier
  type-qualifier-list type-qualifier

parameter-type-list:
  parameter-list
  parameter-list , ...

parameter-list:
  parameter-declaration
  parameter-list , parameter-declaration

parameter-declaration:
  attribute-specifier-sequence_opt declaration-specifiers declarator
  attribute-specifier-sequence_opt declaration-specifiers abstract-declarator_opt

identifier-list:
  identifier
  identifier-list , identifier
\end{verbatim}

Semantics

2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage
duration, and type indicated by the declaration specifiers.

3 A full declarator is a declarator that is not part of another declarator. If, in the nested sequence of declarators in a full declarator, there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be variably modified. Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified.

4 In the following subclauses, consider a declaration

\[ T \; D_1 \]

where \( T \) contains the declaration specifiers that specify a type \( T \) (such as \texttt{int}) and \( D_1 \) is a declarator that contains an identifier \texttt{id}ent. The type specified for the identifier \texttt{id}ent in the various forms of declarator is described inductively using this notation.

5 If, in the declaration “\( T \; D_1 \)”, \( D_1 \) has the form

\[ \text{identifier} \; \text{attribute-specifier-sequence}_{\text{opt}} \]

then the type specified for \texttt{id}ent is \( T \) and the optional attribute specifier sequence appertains to \( D_1 \).

6 If, in the declaration “\( T \; D_1 \)”, \( D_1 \) has the form

\[ ( \; D \) \]

then \texttt{id}ent has the type specified by the declaration “\( T \; D \)”. Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Implementation limits

7 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or \texttt{void} type, either directly or via one or more \texttt{typedef}s.

Forward references: array declarators (6.7.6.2), type definitions (6.7.8).

6.7.6.1 Pointer declarators

Semantics

1 If, in the declaration “\( T \; D_1 \)”, \( D_1 \) has the form

\[ * \; \text{attribute-specifier-sequence}_{\text{opt}} \; \text{type-qualifier-list}_{\text{opt}} \; D \]

and the type specified for \texttt{id}ent in the declaration “\( T \; D \)" is “\texttt{derived-declarator-type-list} \( T \)”, then the type specified for \texttt{id}ent is “\texttt{derived-declarator-type-list} type-qualifier-list pointer to \( T \)". For each type qualifier in the list, \texttt{id}ent is a so-qualified pointer. The optional attribute specifier sequence appertains to the pointer and not the object pointed to.

2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.

3 \textbf{EXAMPLE} The following pair of declarations demonstrates the difference between a “variable pointer to a constant value” and a “constant pointer to a variable value”.

\begin{verbatim}
const int *ptr_to_constant;
int *const constant_ptr;
\end{verbatim}

The contents of any object pointed to by \texttt{ptr_to_constant} cannot be modified through that pointer, but \texttt{ptr_to_constant} itself can be changed to point to another object. Similarly, the contents of the \texttt{int} pointed to by \texttt{constant_ptr} can be modified, but \texttt{constant_ptr} itself always points to the same location.

4 The declaration of the constant pointer \texttt{constant_ptr} can be clarified by including a definition for the type “\texttt{pointer to int}”.

\begin{verbatim}
typedef int *int_ptr;
const int_ptr constant_ptr;
\end{verbatim}

declares \texttt{constant_ptr} as an object that has type “\texttt{const-qualified pointer to int}”.

§ 6.7.6.1 Language 105
6.7.6.2 Array declarators

Constraints

1. In addition to optional type qualifiers and the keyword `static`, the `[ ]` may delimit an expression or `*`. If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword `static` shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

2. If an identifier is declared as having a variably modified type, it shall be an ordinary identifier (as defined in 6.2.3), have no linkage, and have either block scope or function prototype scope. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

Semantics

3. If, in the declaration `"T D1"`, `D1` has one of the forms:
   
   ```
   D [ type-qualifier-list_opt assignment-expression_opt ] attribute-specifier-sequence_opt
   D [ static type-qualifier-list_opt assignment-expression ] attribute-specifier-sequence_opt
   D [ type-qualifier-list static assignment-expression ] attribute-specifier-sequence_opt
   D [ type-qualifier-list_opt * ] attribute-specifier-sequence_opt.
   ```

   and the type specified for `ident` in the declaration `"T D"` is "derived-declarator-type-list T", then the type specified for `ident` is "derived-declarator-type-list array of T".\(^{151}\) The optional attribute specifier sequence appertains to the array. (See 6.7.6.3 for the meaning of the optional type qualifiers and the keyword `static`.)

4. If the size is not present, the array type is an incomplete type. If the size is `*` instead of being an expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations or type names with function prototype scope;\(^{152}\) such arrays are nonetheless complete types. If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a *variable length array* type. (Variable length arrays are a conditional feature that implementations need not support; see 6.10.8.3.)

5. If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by `*`; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. Where a size expression is part of the operand of a `_Alignof` operator, that expression is not evaluated.

6. For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

7. **EXAMPLE 1**

   ```
   float fa[11], *afp[17];
   ```

   declares an array of `float` numbers and an array of pointers to `float` numbers.

8. **EXAMPLE 2** Note the distinction between the declarations

   ```
   extern int *x;
   extern int y[];
   ```

---

\(^{151}\)When several “array of” specifications are adjacent, a multidimensional array is declared.

\(^{152}\)Thus, `*` can be used only in function declarations that are not definitions (see 6.7.6.3).
The first declares \( x \) to be a pointer to \( \text{int} \), the second declares \( y \) to be an array of \( \text{int} \) of unspecified size (an incomplete type), the storage for which is defined elsewhere.

### Example 3
The following declarations demonstrate the compatibility rules for variably modified types.

```c
extern int n;
extern int m;

void fcompat(void)
{
   int a[n][6][m];
   int (*p)[4][n+1];
   int c[n][n][6][m];
   int (*r)[n][n][n+1];
   p = a; // invalid: not compatible because 4 != 6
   r = c; // compatible, but defined behavior only if
           // n == 6 and m == n+1
}
```

### Example 4
All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the \_Thread_local, static, or extern\ storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the static storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```c
extern int n;
int A[n]; // invalid: file scope VLA
extern int (*p2)[n]; // invalid: file scope VM
int B[100]; // valid: file scope but not VM

void fVla(int m, int C[m][m])); // valid: VLA with prototype scope
void fVLa(int m, int C[m][m]) // valid: adjusted to auto pointer to VLA
{
   typedef int VLA[m][m]; // valid: block scope typedef VLA
   struct tag {
      int (*y)[n]; // invalid: y not ordinary identifier
      int z[n]; // invalid: z not ordinary identifier
   };
   int D[m]; // valid: auto VLA
   static int E[m]; // invalid: static block scope VLA
   extern int F[m]; // invalid: F has linkage and is VLA
   int (*s)[m]; // valid: auto pointer to VLA
   extern int (*r)[m]; // invalid: r has linkage and points to VLA
   static int (*q)[m] = &B; // valid: q is a static block pointer to VLA
}
```

Forward references: function declarators (6.7.6.3), function definitions (6.9.1), initialization (6.7.9).

#### 6.7.6.3 Function declarators

**Constraints**

1. A function declarator shall not specify a return type that is a function type or an array type.
2. The only storage-class specifier that shall occur in a parameter declaration is register.
3. An identifier list in a function declarator that is not part of a definition of that function shall be empty.

After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.
Semantics

1. If, in the declaration “\texttt{T D1}”, \texttt{D1} has the form

   \begin{verbatim}
   D ( parameter-type-list, pt ) attribute-specifier-sequenceopt
   or
   D ( identifier-listopt )
   \end{verbatim}

   and the type specified for \texttt{ident} in the declaration “\texttt{T D}” is “derived-declarator-type-list \texttt{T}”, then the type specified for \texttt{ident} is “derived-declarator-type-list function returning the unqualified version of \texttt{T}”. \textbf{The optional attribute specifier sequence appertains to the function type.}

2. A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.

3. A declaration of a parameter as “array of type” shall be adjusted to “qualified pointer to type”, where the type qualifiers (if any) are those specified within the \{ and \} of the array type derivation. If the keyword \texttt{static} also appears within the \{ and \} of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

4. A declaration of a parameter as “function returning type” shall be adjusted to “pointer to function returning type”, as in 6.3.2.1.

5. If the list terminates with an ellipsis (\texttt{\ldots}), no information about the number or types of the parameters after the comma is supplied.\footnote{The macros defined in the <\texttt{stdarg.h}> header (7.16) can be used to access arguments that correspond to the ellipsis.}

6. The special case of an unnamed parameter of type \texttt{void} as the only item in the list specifies that the function has no parameters.

7. If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

8. If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the \{ \* \} notation in their sequences of declarator specifiers to specify variable length array types.

9. The \texttt{storage class} specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. \textbf{The optional attribute specifier sequence in a parameter declaration appertains to the parameter.}

10. An identifier list declares only the identifiers of the parameters of the function. An empty list in \texttt{For} a function declarator that \textit{without a parameter type list: if it is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function and the effect is as if it were declared with a parameter type list consisting of the keyword \texttt{void}; otherwise it specifies that no information about the number or types of the parameters is supplied.} A function declarator provides a \textit{prototype} for the function if it includes a parameter type list.\footnote{See “future language directions” (6.11.6).} Otherwise, a function declaration is said to have no prototype.

11. For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that \textit{has none and is not part of a function definition and that contains an empty identifier list}, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. In the determination of type compatibility and of a

\footnote{This implies that a function definition without a parameter list provides a prototype, and that subsequent calls to that function in the same translation unit are constrained not to provide any argument to the function call. Thus a definition of a function without parameter list and one that has such a list consisting of the keyword \texttt{void} are fully equivalent.}
composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

15 **EXAMPLE 1** The declaration

```c
int f(void), *fip(), (*pfi());
```

declares a function `f` with no parameters returning an `int`, a function `fip` with no parameter specification returning a pointer to an `int`, and a pointer `pfi` to a function with no parameter specification returning an `int`. It is especially useful to compare the last two. The binding of `*fip()` is `*fip()`, so that the declaration suggests, and the same construction in an expression requires, the calling of a function `fip`, and then using indirection through the pointer result to yield an `int`. In the declaration `(*pfi)()`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an `int`.

16 If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions `f` and `fip` have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer `pfi` has block scope and no linkage.

17 **EXAMPLE 2** The declaration

```c
int (*apfi[][3])(int *x, int *y);
```

declares an array `apfi` of three pointers to functions returning `int`. Each of these functions has two parameters that are pointers to `int`. The identifiers `x` and `y` are declared for descriptive purposes only and go out of scope at the end of the declaration of `apfi`.

18 **EXAMPLE 3** The declaration

```c
int (*fpfi(int (*)(*) (long), int))(int, ...);
```

declares a function `fpfi` that returns a pointer to a function returning an `int`. The function `fpfi` has two parameters: a pointer to a function returning an `int` (with one parameter of type `long`), and an `int`. The pointer returned by `fpfi` points to a function that has one `int` parameter and accepts zero or more additional arguments of any type.

19 **EXAMPLE 4** The following prototype has a variably modified parameter.

```c
void addscalar(int n, int m,
    double a[n][n*m+300], double x);
```

```c
int main()
{
    double b[4][308];
    addscalar(4, 2, b, 2.17);
    return 0;
}
```

```c
void addscalar(int n, int m,
    double a[n][n*m+300], double x)
{
    for (int i = 0; i < n; i++)
        for (int j = 0, k = n*m+300; j < k; j++)
            // a is a pointer to a VLA with n*m+300 elements
            a[i][j] += x;
}
```

20 **EXAMPLE 5** The following are all compatible function prototype declarators.

```c
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[+][+]);
double maximum(int n, int m, double a[][+]);
double maximum(int n, int m, double a[ ][+]);
```

as are:

```c
void f(double (* restrict a)[5]);
```
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);

(Note that the last declaration also specifies that the argument corresponding to \(a\) in any call to \(f\) can be expected to be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Forward references: function definitions (6.9.1), type names (6.7.7).

6.7.7 Type names

Syntax

\[
\begin{align*}
type-name: & \quad \text{specifier-qualifier-list abstract-declarator}_{opt} \\
abstract-declarator: & \quad \text{pointer} \\
pointer_{opt} direct-abstract-declarator \\
direct-abstract-declarator: & \quad ( \text{abstract-declarator} ) \\
\quad \text{direct-abstract-declarator}_{opt} [ \text{type-qualifier-list}_{opt} ] \\
attribute-specifier-sequence_{opt} \\
array-abstract-declarator: & \quad \text{function-abstract-declarator}_{opt} \quad \text{attribute-specifier-sequence}_{opt} \\
\quad \text{direct-abstract-declarator}_{opt} [- \text{static} \text{type-qualifier-list}_{opt} ] \\
\quad \quad assignment-expression_{opt} ] \\
\quad \text{direct-abstract-declarator}_{opt} [ \text{type-qualifier-list} \text{static} ] \\
\quad \quad assignment-expression ] \\
\quad \text{direct-abstract-declarator}_{opt} [ * ] \\
function-abstract-declarator: & \quad \text{direct-abstract-declarator}_{opt} ( \text{parameter-type-list}_{opt} ) \\
\end{align*}
\]

Semantics

In several contexts, it is necessary to specify a type. This is accomplished using a type name, which is syntactically a declaration for a function or an object of that type that omits the identifier.\(^{156}\)

The optional attribute specifier sequence in a direct abstract declarator appertains to the preceding array or function type. The attribute specifier sequence affects the type only for the declaration it appears in, not other declarations involving the same type.

EXAMPLE The constructions

\[
\begin{align*}
(a) & \quad \text{int} \\
(b) & \quad \text{int} * \\
(c) & \quad \text{int} *[3] \\
(d) & \quad \text{int} (*)([3] \\
(e) & \quad \text{int} (*)(*)[3] \\
(f) & \quad \text{int} *() \\
(g) & \quad \text{int} (*)(void) \\
(h) & \quad \text{int} (*const [])(unsigned \text{int}, \ldots)
\end{align*}
\]

name respectively the types (a) int, (b) pointer to int, (c) array of three pointers to int, (d) pointer to an array of three int s, (e) pointer to a variable length array of an unspecified number of int s, (f) function with no parameter specification returning a pointer to int, (g) pointer to function with no parameters returning an int, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type unsigned int and an unspecified number of other parameters, returning an int.

\(^{156}\)As indicated by the syntax, empty parentheses in a type name are interpreted as “function with no parameter specification”, rather than redundant parentheses around the omitted identifier.
6.7.8 Type definitions

Syntax

1  typedef-name:
     identifier

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope.

Semantics

3 In a declaration whose storage-class specifier is typedef, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.6. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A typedef declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

   typedef T type_ident;
type_ident D;

type_ident is defined as a typedef name with the type specified by the declaration specifiers in T (known as T), and the identifier in D has the type “derived-declarator-type-list T” where the derived-declarator-type-list is specified by the declarators of D. A typedef name shares the same name space as other identifiers declared in ordinary declarators.

4 EXAMPLE 1 After

   typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;

the constructions

   MILES distance;
   extern KLICKSP *metricp;
   range x;
   range z, *zp;

are all valid declarations. The type of distance is int, that of metricp is “pointer to function with no parameter specification returning int”, and that of x and z is the specified structure; zp is a pointer to such a structure. The object distance has a type compatible with any other int object.

5 EXAMPLE 2 After the declarations

   typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;

type t1 and the type pointed to by tp1 are compatible. Type t1 is also compatible with type struct s1, but not compatible with the types struct s2, t2, the type pointed to by tp2, or int.

6 EXAMPLE 3 The following obscure constructions

   typedef signed int t;
typedef int plain;
struct tag {
   unsigned t:4;
   const t:5;
   plain r:5;
};
declare a typedef name t with type signed int, a typedef name plain with type int, and a structure with three bit-field members, one named t that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it could
be accessed) would contain values in either the range \([-15, +15]\) or \([-16, +15]\), and one named \(r\) that contains values in one of the ranges \([0, 31]\), \([-15, +15]\), or \([-16, +15]\). (The choice of range is implementation-defined.) The first two bit-field declarations differ in that \texttt{unsigned}\ is a type specifier (which forces \(t\) to be the name of a structure member), while \texttt{const}\ is a type qualifier (which modifies \(t\) which is still visible as a typedef name). If these declarations are followed in an inner scope by

\[
    t\ f(t\ t());
    \text{long\ } t;
\]

then a function \(f\) is declared with type “function returning \texttt{signed int}\” with one unnamed parameter with type pointer to function returning \texttt{signed int}\ with one unnamed parameter with type \texttt{signed int}\”, and an identifier \(t\) with type \texttt{long\ int}.

\section*{Example 4}

On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the \texttt{signal}\ function specify exactly the same type, the first without making use of any typedef names.

\begin{verbatim}
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
\end{verbatim}

\section*{Example 5}

If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

\begin{verbatim}
void copyt(int n)
{
    typedef int B[n]; // B is n ints, n evaluated now
    n += 1;
    B a;              // a is n ints, n without += 1
    int b[n];         // a and b are different sizes
    for (int i = 1; i < n; i++)
        a[i-1] = b[i];
}
\end{verbatim}

\section*{6.7.9 Initialization}

\subsection*{Syntax}

\begin{verbatim}
initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }

initializer-list:
    designation_opt initializer
    initializer-list , designation_opt initializer

designation:
    designator-list =

designator-list:
    designator
    designator-list designator

designator:
    [ constant-expression ]
    . identifier
\end{verbatim}

\subsection*{Constraints}

No initializer shall attempt to provide a value for an object not contained within the entity being initialized.
The type of the entity to be initialized shall be an array of unknown size or a complete object type that is not a variable length array type.

All the expressions in an initializer for an object that has static or thread storage duration shall be constant expressions or string literals.

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

If a designator has the form

```
[ constant-expression ]
```

then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.

If a designator has the form

```
. identifier
```

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

### Semantics

An initializer specifies the initial value stored in an object.

Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.

If an object that has static or thread storage duration is not initialized explicitly, then:

- if it has pointer type, it is initialized to a null pointer;
- if it has arithmetic type, it is initialized to (positive or unsigned) zero;
- if it is an aggregate, every member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;
- if it is a union, the first named member is initialized (recursively) according to these rules, and any padding is initialized to zero bits;

The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

The rest of this subclause deals with initializers for objects that have aggregate or union type.

The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.

An array of character type may be initialized by a character string literal or UTF–8 string literal, optionally enclosed in braces. Successive bytes of the string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

An array with element type compatible with a qualified or unqualified version of `wchar_t`, `char16_t`, or `char32_t` may be initialized by a wide string literal with the corresponding encoding prefix (`L`, `u`, or `U`, respectively), optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.

Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.
Each brace-enclosed initializer list has an associated current object. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.\(^{157}\) In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization then continues forward in order, beginning with the next subobject after that described by the designator.\(^{158}\)

Each designator list begins its description with the current object associated with the closest surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.\(^{159}\) The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.

The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;\(^{160}\) all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.

If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.

If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. The array type is completed at the end of its initializer list.

The evaluations of the initialization list expressions are indeterminately sequenced with respect to one another and thus the order in which any side effects occur is unspecified.\(^{161}\)

**EXAMPLE 1** Provided that `<complex.h>` has been `#included`, the declarations

```c
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize `i` with the value 3 and `c` with the value $5.0 + i3.0$.

**EXAMPLE 2** The declaration

```c
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

**EXAMPLE 3** The declaration

```c
int y[4][3] = {
  { 1, 3, 5 },
  { 2, 4, 6 },
};
```

\(^{157}\) If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

\(^{158}\) After a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

\(^{159}\) Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

\(^{160}\) Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

\(^{161}\) In particular, the evaluation order need not be the same as the order of subobject initialization.
is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of \( y \) (the array object \( y[0] \)), namely \( y[0][0], y[0][1], \) and \( y[0][2] \). Likewise the next two lines initialize \( y[1] \) and \( y[2] \). The initializer ends early, so \( y[3] \) is initialized with zeros. Precisely the same effect could have been achieved by

```c
int y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for \( y[0] \) does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for \( y[1] \) and \( y[2] \).

**EXAMPLE 4** The declaration

```c
int z[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of \( z \) as specified and initializes the rest with zeros.

**EXAMPLE 5** The declaration

```c
struct {
    int a[3], b;
} w[] = { { 1 }, 2);
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: \( w[0].a[0] \) is 1 and \( w[1].a[0] \) is 2; all the other elements are zero.

**EXAMPLE 6** The declaration

```c
short q[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: \( q[0][0][0] \) is 1, \( q[1][0][0] \) is 2, \( q[1][0][1] \) is 3, and 4, 5, and 6 initialize \( q[2][0][0], q[2][0][1], \) and \( q[2][1][0] \), respectively; all the rest are zero. The initializer for \( q[0][0] \) does not begin with a left brace, so up to six items from the current list could be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for \( q[1][0] \) and \( q[2][0] \) do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```c
short q[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};
```

or by:

```c
short q[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5 },
    { 6 }
};
```
in a fully bracketed form.

30 Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

31 **EXAMPLE 7** One form of initialization that completes array types involves typedef names. Given the declaration

```c
typedef int A[]; // OK - declared with block scope
```

the declaration

```c
A a = { 1, 2 }, b = { 3, 4, 5 };
```

is identical to

```c
int a[] = { 1, 2 }, b[] = { 3, 4, 5 };
```

due to the rules for incomplete types.

32 **EXAMPLE 8** The declaration

```c
char s[] = "abc", t[3] = "abc";
```

defines “plain” char array objects `s` and `t` whose elements are initialized with character string literals. This declaration is identical to

```c
char s[] = { 'a', 'b', 'c', '\0' },
t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```c
char *p = "abc";
```

defines `p` with type “pointer to char” and initializes it to point to an object with type “array of char” with length 4 whose elements are initialized with a character string literal. If an attempt is made to use `p` to modify the contents of the array, the behavior is undefined.

33 **EXAMPLE 9** Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```c
enum { member_one, member_two };
const char *nm[] = {
    [member_two] = "member two",
    [member_one] = "member one",
};
```

34 **EXAMPLE 10** Structure members can be initialized to nonzero values without depending on their order:

```c
div_t answer = {.quot = 2, .rem = -1 };
```

35 **EXAMPLE 11** Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```c
struct { int a[3], b; } w[] =
    { [0].a = {1}, [1].a[0] = 2 };
```

36 **EXAMPLE 12**

```c
struct T {
    int k;
    int l;
};

struct S {
    int i;
    struct T t;
};
```
The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, which is specified in the implementation-defined <stddef.h> (7.19).

### 6.7.10 Static assertions

#### Syntax

```c
_Static_assert ( constant-expression, string-literal ) ;
_Static_assert ( constant-expression ) ;
```

#### Constraints

1. The constant expression shall compare unequal to 0.

#### Semantics

1. The constant expression shall be an integer constant expression. If the value of the constant expression compares unequal to 0, the declaration has no effect. Otherwise, the constraint is violated and the implementation shall produce a diagnostic message that includes the text of the string literal, if present, except that characters not in the basic source character set are not required to appear in the message.

#### Forward references: diagnostics (7.2).

### 6.7.11 Attributes

1. Attributes specify additional information for various source constructs such as types, variables, identifiers, or blocks. They are identified by an attribute token, which can either be a **attribute prefixed token** (for implementation-specific attributes) or a **standard attribute** specified by an identifier (for attributes specified in this document).

2. Support for any of the standard attributes specified in this document is implementation-defined and optional. For an attribute token (including an attribute prefixed token) not specified in this document, the behavior is implementation-defined. Any attribute token that is not supported by the implementation is ignored.

3. Attributes are said to appertain to some source construct, identified by the syntactic context where they appear, and for each individual attribute, the corresponding clause constrains the syntactic context in which this appertainance is valid. The attribute specifier sequence appertaining to some source construct shall contain only attributes that are allowed to apply to that source construct.

4. In all aspects of the language, a standard attribute specified by this document as an identifier `attr` and an identifier of the form `attr__` shall behave the same when used as an attribute token,
except for the spelling.\footnote{\textcopyright 182)}

**Recommended practice**

5 It is recommended that implementations support all standard attributes as defined in this document.

6.7.11.1 General

**Syntax**

\[
\begin{align*}
\text{attribute-specifier-sequence:} & \quad \text{attribute-specifier-sequence}_{\text{opt}}, \text{attribute-specifier} \\
\text{attribute-specifier:} & \quad [ \{ \text{attribute-list} \} ] \\
\text{attribute-list:} & \quad \text{attribute} \_ {\text{opt}} \\
\text{attribute:} & \quad \text{attribute-list} \_ {\text{opt}}, \text{attribute}_{\text{opt}} \\
\text{attribute-token:} & \quad \text{standard-attribute} \\
\text{attribute-argument-clause:} & \quad \text{attribute-token} \_ {\text{opt}}, \text{attribute-argument-clause}_{\text{opt}} \\
\text{standard-attribute:} & \quad \text{attribute-prefixed-token} \\
\text{attribute-prefixed-token:} & \quad \text{attribute-prefix} :: \text{identifier} \\
\text{attribute-prefix:} & \quad \text{identifier} \\
\text{attribute-argument-clause:} & \quad \{ \text{balanced-token-sequence}_{\text{opt}} \} \\
\text{balanced-token-sequence:} & \quad \text{balanced-token} \\
\text{balanced-token:} & \quad \{ \text{balanced-token-sequence}_{\text{opt}} \} \\
\text{balanced-token-sequence:} & \quad \{ \text{balanced-token} \} \\
\text{balanced-token:} & \quad \{ \text{balanced-token-sequence}_{\text{opt}} \} \\
\end{align*}
\]

**Constraints**

2 The identifier in a standard attribute shall be one of:

\[
\begin{align*}
\text{deprecated} & \quad \text{maybe_unused} \\
\text{fallthrough} & \quad \text{nodiscard}
\end{align*}
\]

**Semantics**

3 An attribute specifier that contains no attributes has no effect. The order in which attribute tokens appear in an attribute list is not significant. If a keyword (6.4.1) that satisfies the syntactic requirements of an identifier (6.4.2) is contained in an attribute token, it is considered an identifier. A strictly conforming program using a standard attribute remains strictly conforming in the absence of that attribute.\footnote{\textcopyright 183)}

4 **NOTE** For each standard attribute, the form of the balanced token sequence, if any, will be specified.

\footnote{\textcopyright 182) Thus, the attributes \{\text{nodiscard}\} and \{\text{--nodiscard--}\} can be freely interchanged. Implementations are encouraged to behave similarly for attribute tokens (including attribute prefixed tokens) they provide.}

\footnote{\textcopyright 183) Standard attributes specified by this document can be parsed but ignored by an implementation without changing the semantics of a correct program; the same is not true for attributes not specified by this document.}
Recommended Practice

5 Each implementation should choose a distinctive name for the attribute prefix in an attribute prefixed token. Implementations should not define attributes without an attribute prefix unless it is a standard attribute as specified in this document.

6 EXAMPLE 1 Suppose that an implementation chooses the attribute prefix hal and provides specific attributes named daisy and rosie.

```c
[[deprecated, hal::daisy]] double nine1000(double);
[[deprecated]] [[hal::daisy]] double nine1000(double);
[[deprecated]] double nine1000 [[hal::daisy]] (double);
```

Then all the following declarations should be equivalent aside from the spelling:

```c
[[__deprecated__, __hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] [[__hal__::__daisy__]] double nine1000(double);
[[__deprecated__]] double nine1000 [[__hal__::__daisy__]] (double);
```

These use the alternate spelling that is required for all standard attributes and recommended for prefixed attributes. These may be better-suited for use in header files, where the use of the alternate spelling avoids naming conflicts with user-provided macros.

7 EXAMPLE 2 For the same implementation, the following two declarations are equivalent, because the ordering inside attribute lists is not important.

```c
[[hal::daisy, hal::rosie]] double nine999(double);
[[hal::rosie, hal::daisy]] double nine999(double);
```

On the other hand the following two declarations are not equivalent, because the ordering of different attribute specifiers may affect the semantics.

```c
[[hal::daisy]] [[hal::rosie]] double nine999(double);
[[hal::rosie]] [[hal::daisy]] double nine999(double); // may have different semantics
```

6.7.11.2 The noexcept attribute

Constraint

1 The noexcept attribute shall be applied to the identifier in a function declarator or to the definition of a structure, union, or enumeration type. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics

2 A name or entity declared without the noexcept attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.

Recommended Practice

3 A noexcept call is a function call expression that calls a function previously declared with attribute noexcept, or whose return type is a structure, union, or enumeration type marked with attribute noexcept. Evaluation of a noexcept call as a void expression (6.8.3) is discouraged unless explicitly cast to void. Implementations are encouraged to issue a diagnostic in such cases. This is typically because immediately discarding the return value of a noexcept call has surprising consequences.

4 EXAMPLE 1

```c
struct [[noexcept]] error_info { /*...*/ };  
struct error_info enable_missile_safety_mode(void);  
void launch_missiles(void);  
void test_missiles(void) {  
enable_missile_safety_mode();  
launch_missiles();  }
```
EXAMPLE 2

```c
[[nodiscard]] int important_func(void);
void call(void) {
  int i = important_func();
}
```

No diagnostic for the call to `important_func` is encouraged despite the value of `i` not being used.

6.7.11.3 The `maybe_unused` attribute

Constraint

1. The `maybe_unused` attribute shall be applied to the declaration of a structure, a union, a `typedef` name, a variable, a structure or union member, a function, an enumeration, or an enumerator. It shall appear at most once in each attribute list and no attribute argument clause shall be present.

Semantics

2. The `maybe_unused` attribute indicates that a name or entity is possibly intentionally unused. A name or entity declared without the `maybe_unused` attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

Recommended Practice

3. For an entity marked `maybe_unused`, implementations are encouraged not to emit a diagnostic that the entity is unused, or that the entity is used despite the presence of the attribute.

EXAMPLE

```c
[[maybe_unused]] void f([[maybe_unused]] int i) {
  [[maybe_unused]] int j = i + 100;
  assert(j);
}
```

Implementations are encouraged not to diagnose that `j` is unused, whether or not `NDEBUG` is defined.

6.7.11.4 The `deprecated` attribute

Constraint

1. The `deprecated` attribute shall be applied to the declaration of a structure, a union, a `typedef` name, a variable, a structure or union member, a function, an enumeration, or an enumerator. It shall appear at most once in each attribute list.

2. If an attribute argument clause is present, it shall have the form:

```
~~~~~~(string-literal)
```

Semantics

3. The `deprecated` attribute can be used to mark names and entities whose use is still allowed, but is discouraged for some reason.

4. A name or entity declared without the `deprecated` attribute can later be redeclared with the attribute and vice versa. An entity is considered marked with the attribute after the first declaration that marks it.

Recommended Practice

5. Implementations should use the `deprecated` attribute to produce a diagnostic message in case the program refers to a name or entity other than to declare it, after a declaration that specifies the attribute, when the reference to the name or entity is not within the context of a related deprecated

---

\[164\] In particular, `deprecated` is appropriate for names and entities that are obsolescent, insecure, unsafe, or otherwise unfit for purpose.
entity. The diagnostic message may include text provided by the string literal within the attribute argument clause of any deprecated attribute applied to the name or entity.

EXAMPLE

```c
struct [[deprecated]] S {
    int a;
};

enum [[deprecated]] E1 {
    one
};
(enum E2 {
    two [[deprecated("use 'three' instead")]],
    three
};
[[deprecated]] typedef int Foo;

void f1(struct S s) { // Diagnose use of S
    int i = one; // Diagnose use of E1
    int j = two; // Diagnose use of two: "use 'three' instead"
    int k = three;
    Foo f; // Diagnose use of Foo
}

[[deprecated]] void f2(struct S s) {
    int i = one;
    int j = two;
    int k = three;
    Foo f;
}

struct [[deprecated]] T {
    Foo f;
    struct S s;
};
```

Implementations are encouraged to diagnose the use of deprecated entities within a context which is not itself deprecated, as indicated for function f1, but not to diagnose within function f2 and struct T, as they are themselves deprecated.

6.7.11.5 The fallthrough attribute

Constraint

1 The attribute token fallthrough shall only appear in an attribute declaration (6.7); such a declaration is a fallthrough declaration. The attribute token fallthrough shall appear at most once in each attribute list and no attribute argument clause shall be present. A fallthrough declaration may only appear within an enclosing switch statement (6.8.4.2). The next statement that would be executed after a fallthrough declaration shall be a labeled statement whose label is a case label or default label for the same switch statement.

Recommended Practice

2 The use of a fallthrough declaration is intended to suppress a diagnostic that an implementation might otherwise issue for a case or default label that is reachable from another case or default label along some path of execution. Implementations are encouraged to issue a diagnostic if a fallthrough declaration is not dynamically reachable.

EXAMPLE

```c
void f(int n) {
    void g(void), h(void), i(void);
    switch (n) {
```
case 1: /* diagnostic on fallthrough discouraged */
case 2:
g();
    [[fallthrough]];
case 3: /* diagnostic on fallthrough discouraged */
h();
case 4: /* fallthrough diagnostic encouraged */
i();
    [[fallthrough]]; /* constraint violation */
}
6.8 Statements and blocks

Syntax

1

statement: labeled-statement
expression-statement
compound-statement
expression-statement
case constant-expression : statement
expression-statement
attribute-specifier-sequence_opt
compound-statement
attribute-specifier-sequence_opt
selection-statement
attribute-specifier-sequence_opt
iteration-statement
attribute-specifier-sequence_opt
jump-statement

Semantics

2 A statement specifies an action to be performed. Except as indicated, statements are executed in sequence.

3 A block allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.

4 A full expression is an expression that is not part of another expression, nor part of a declarator or abstract declarator. There is also an implicit full expression in which the non-constant size expressions for a variably modified type are evaluated; within that full expression, the evaluation of different size expressions are unsequenced with respect to one another. There is a sequence point between the evaluation of a full expression and the evaluation of the next full expression to be evaluated.

5 NOTE Each of the following is a full expression:

— a full declarator for a variably modified type,
— an initializer that is not part of a compound literal,
— the expression in an expression statement,
— the controlling expression of a selection statement (if or switch),
— the controlling expression of a while or do statement,
— each of the (optional) expressions of a for statement,
— the (optional) expression in a return statement.

While a constant expression satisfies the definition of a full expression, evaluating it does not depend on nor produce any side effects, so the sequencing implications of being a full expression are not relevant to a constant expression.

Forward references: expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the return statement (6.8.6.4).

6.8.1 Labeled statements

Syntax

1 labeled-statement:

attribute-specifier-sequence_opt identifier : statement
attribute-specifier-sequence_opt case constant-expression : statement
attribute-specifier-sequence_opt default : statement

Constraints

2 A case or default label shall appear only in a switch statement. Further constraints on such labels are discussed under the switch statement.

3 Label names shall be unique within a function.
Semantics

Any statement may be preceded by a prefix that declares an identifier as a label name. The optional attribute specifier sequence appertains to the label. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Forward references: the goto statement (6.8.6.1), the switch statement (6.8.4.2).

6.8.2 Compound statement

Syntax

```
compound-statement:
  { block-item-list_opt }
```

```
block-item-list:
  block-item
  block-item-list block-item
```

```
block-item:
  declaration
  statement
```

Semantics

A compound statement is a block.

6.8.3 Expression and null statements

Syntax

```
expression-statement:
  expression_opt ;
  attribute-specifier-sequence expression ;
```

Semantics

The attribute specifier sequence appertains to the expression. The expression in an expression statement is evaluated as a void expression for its side effects.\(^{(165)}\)

A null statement (consisting of just a semicolon) performs no operations.

EXAMPLE 1 If a function call is evaluated as an expression statement for its side effects only, the discarding of its value can be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

EXAMPLE 2 In the program fragment

```
char *s;
/* ... */
while (*s++ != ' \0')
  
;  
```

a null statement is used to supply an empty loop body to the iteration statement.

EXAMPLE 3 A null statement can also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
  /* ... */
  while (loop2) {
    /* ... */
    if (want_out)
      goto end_loop1;
    /* ... */
  }
  /* ... */
}  
```

\(^{(165)}\)Such as assignments, and function calls which have side effects.
end_loop1:;
}

Forward references: iteration statements (6.8.5).

6.8.4 Selection statements

Syntax

1

selection-statement:
if ( expression ) statement
if ( expression ) statement else statement
switch ( expression ) statement

Semantics

2
A selection statement selects among a set of statements depending on the value of a controlling expression.

3
A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

6.8.4.1 The if statement

Constraints

1
The controlling expression of an if statement shall have scalar type.

Semantics

2
In both forms, the first substatement is executed if the expression compares unequal to 0. In the else form, the second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

3
An else is associated with the lexically nearest preceding if that is allowed by the syntax.

6.8.4.2 The switch statement

Constraints

1
The controlling expression of a switch statement shall have integer type.

2
If a switch statement has an associated case or default label within the scope of an identifier with a variably modified type, the entire switch statement shall be within the scope of that identifier.\(^{166}\)

3
The expression of each case label shall be an integer constant expression and no two of the case constant expressions in the same switch statement shall have the same value after conversion. There may be at most one default label in a switch statement. (Any enclosed switch statement may have a default label or case constant expressions with values that duplicate case constant expressions in the enclosing switch statement.)

Semantics

4
A switch statement causes control to jump to, into, or past the statement that is the switch body, depending on the value of a controlling expression, and on the presence of a default label and the values of any case labels on or in the switch body. A case or default label is accessible only within the closest enclosing switch statement.

5
The integer promotions are performed on the controlling expression. The constant expression in each case label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched case label. Otherwise, if there is a default label, control jumps to the labeled statement. If no converted case constant expression matches and there is no default label, no part of the switch body is executed.

\(^{166}\)That is, the declaration either precedes the switch statement, or it follows the last case or default label associated with the switch that is in the block containing the declaration.
Implementation limits

6 As discussed in 5.2.4.1, the implementation may limit the number of `case` values in a `switch` statement.
EXAMPLE  In the artificial program fragment

```
switch (expr)
{
    int i = 4;
    f(i);
    case 0:
        i = 17;
        /* falls through into default code */
    default:
        printf("%d\n", i);
}
```

the object whose identifier is i exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the printf function will access an indeterminate value. Similarly, the call to the function f cannot be reached.

6.8.5  Iteration statements

Syntax

1

iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( expression_opt ; expression_opt ; expression_opt ) statement
    for ( declaration expression_opt ; expression_opt ) statement

Constraints

2  The controlling expression of an iteration statement shall have scalar type.

3  The declaration part of a for statement shall only declare identifiers for objects having storage class auto or register.

Semantics

4  An iteration statement causes a statement called the loop body to be executed repeatedly until the controlling expression compares equal to 0. The repetition occurs regardless of whether the loop body is entered from the iteration statement or by a jump.\(^{167}\)

5  An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

6  An iteration statement may be assumed by the implementation to terminate if its controlling expression is not a constant expression,\(^{168}\) and none of the following operations are performed in its body, controlling expression or (in the case of a for statement) its expression-3:\(^{169}\)

   — input/output operations
   — accessing a volatile object
   — synchronization or atomic operations.

6.8.5.1  The while statement

1  The evaluation of the controlling expression takes place before each execution of the loop body.

6.8.5.2  The do statement

1  The evaluation of the controlling expression takes place after each execution of the loop body.

\(^{167}\)Code jumped over is not executed. In particular, the controlling expression of a for or while statement is not evaluated before entering the loop body, nor is clause-1 of a for statement.

\(^{168}\)An omitted controlling expression is replaced by a nonzero constant, which is a constant expression.

\(^{169}\)This is intended to allow compiler transformations such as removal of empty loops even when termination cannot be proven.
### 6.8.5.3 The for statement

The statement

```plaintext
for (clause-1; expression-2; expression-3) statement
```

behaves as follows: The expression `expression-2` is the controlling expression that is evaluated before each execution of the loop body. The expression `expression-3` is evaluated as a void expression after each execution of the loop body. If `clause-1` is a declaration, the scope of any identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If `clause-1` is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.\(^{170}\)

Both `clause-1` and `expression-3` can be omitted. An omitted `expression-2` is replaced by a nonzero constant.

### 6.8.6 Jump statements

#### Syntax

```plaintext
jump-statement:
  goto identifier ;
  continue ;
  break ;
  return expression_opt ;
```

#### Semantics

A jump statement causes an unconditional jump to another place.

#### 6.8.6.1 The goto statement

**Constraints**

The identifier in a `goto` statement shall name a label located somewhere in the enclosing function. A `goto` statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

**Semantics**

A `goto` statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.

**Example 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:

1. The general initialization code accesses objects only visible to the current function.
2. The general initialization code is too large to warrant duplication.
3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by `continue` statements, for example.)

```plaintext
/* ... */
goto first_time;
for (;;) {
  // determine next operation
  /* ... */
  if (need to reinitialize) {
    // reinitialize-only code
    /* ... */
    first_time:
    // general initialization code
```

\(^{170}\)Thus, `clause-1` specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, `expression-2`, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and `expression-3` specifies an operation (such as incrementing) that is performed after each iteration.
EXAMPLE 2 A `goto` statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```c
goto lab3; // invalid: going INTO scope of VLA.
{
    double a[n];
    a[j] = 4.4;
lab3:
    a[j] = 3.3;
goto lab4; // valid: going WITHIN scope of VLA.
    a[j] = 5.5;
lab4:
    a[j] = 6.6;
}
goto lab4; // invalid: going INTO scope of VLA.
```

6.8.6.2 The `continue` statement

Constraints
1. A `continue` statement shall appear only in or as a loop body.

Semantics
2. A `continue` statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```c
while /* ... */ {
    /* ... */
continue;
/* ... */
contin;
}
```

unless the `continue` statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to `goto contin;`\(^{(171)}\)

6.8.6.3 The `break` statement

Constraints
1. A `break` statement shall appear only in or as a switch body or loop body.

Semantics
2. A `break` statement terminates execution of the smallest enclosing `switch` or iteration statement.

6.8.6.4 The `return` statement

Constraints
1. A `return` statement with an expression shall not appear in a function whose return type is `void`. A `return` statement without an expression shall only appear in a function whose return type is `void`.

Semantics
2. A `return` statement terminates execution of the current function and returns control to its caller. A function may have any number of `return` statements.

\(^{(171)}\)Following the `contin:` label is a null statement.
If a `return` statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.\(^{172}\)

**EXAMPLE** In:

```c
struct s { double i; } f(void);
union {
    struct {
        int f1;
        struct s f2;
    } u1;
    struct {
        struct s f3;
        int f4;
    } u2;
} g;

struct s f(void)
{
    return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

\(^{172}\)The `return` statement is not an assignment. The overlap restriction of 6.5.16.1 does not apply to the case of function return. The representation of floating-point values can have wider range or precision than implied by the type; a cast can be used to remove this extra range and precision.
6.9 External definitions

Syntax

1 translation-unit:
   external-declaration
   translation-unit external-declaration

   external-declaration:
      function-definition
      declaration

Constraints

2 The storage-class specifiers auto and register shall not appear in the declaration specifiers in an external declaration.

3 There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a sizeof or _Alignof operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

Semantics

4 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as “external” because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

5 An external definition is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a sizeof or _Alignof operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.\footnote{Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.}

6.9.1 Function definitions

Syntax

1 function-definition:
   declaration-specifiers declarator declaration-list opt compound-statement
   attribute-specifier-sequence opt declaration-specifiers declarator function-body

   declaration-list:
      declaration

   function-body:
      declaration list declaration compound-statement

Constraints

2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.

3 The return type of a function shall be\ void\ or a complete object type other than array type.

4 The storage-class specifier, if any, in the declaration specifiers shall be either\ extern\ or\ static.

5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type\ void,\ in which case there shall not be an identifier.\ No declaration list shall follow.

\footnote{Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.}
If the declarator includes an identifier list, each declaration in the declaration list shall have at least
one declarator; those declarators shall declare only identifiers from the identifier list, and every
identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not
be redeclared as a parameter. The declarations in the declaration list shall contain no storage class
specifier other than register and no initializations.

Semantics

The optional attribute specifier sequence in a function definition appertains to the function.

The declarator in a function definition specifies the name of the function being defined and the
identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies
the and types of all the parameters; each the declarator also serves as a function prototype for later
calls to the same function in the same translation unit. If the declarator includes an identifier list, the
types of the parameters shall be declared in a following declaration list. In either case, the The
type of each parameter is adjusted as described in 7 for a parameter type list 6.7.6.3; the resulting
type shall be a complete object type.

If a function that accepts a variable number of arguments is defined without a parameter type list
that ends with the ellipsis notation, the behavior is undefined.

Each parameter has automatic storage duration; its identifier is an lvalue. The layout of the
storage for parameters is unspecified.

On entry to the function, the size expressions of each variably modified parameter are evaluated
and the value of each argument expression is converted to the type of the corresponding parameter
as if by assignment. (Array expressions and function designators as arguments were converted to
pointers before the call.)

After all parameters have been assigned, the compound statement that constitutes the body of the
function definition body is executed.

Unless otherwise specified, if the } that terminates a function-the function body is reached, and the
value of the function call is used by the caller, the behavior is undefined.

NOTE In a function definition, the type of the function and its prototype cannot be inherited from a typedef:

```
typedef int F(void);
    // type F is “function with no parameters
    // returning int”
F f, g;
F f { /* ... */ }                   // f and g both have type compatible with F
F g() { /* ... */ }                // WRONG: syntax/constraint error
 int f(void) { /* ... */ }          // RIGHT: f has type compatible with F
int g() { /* ... */ }              // RIGHT: g has type compatible with F
F e(void) { /* ... */ }            // e returns a pointer to a function
F e(void) { /* ... */ }            // same: parentheses irrelevant
int (*fp)(void);                   // fp points to a function that has type F
F *fp;
```

EXAMPLE 1 In the following:

```
extern int max(int a, int b)
{
    return a > b ? a: b;
}
```

```
extern is the storage-class specifier and int is the type specifier; max(int a, int b) is the function declarator; and

{ return a > b ? a: b; }
```

173) The intent is that the type category in a function definition cannot be inherited from a typedef.
174) See “future language direction” (22).

174) A parameter identifier cannot be redeclared in the function body except in an enclosed block.
is the function body. The following similar definition uses the identifier list form for the parameter declarations. Here-

\[ \text{int a, b;} \]

is the declaration list for the parameters. The difference between these two definitions is that the first form acts

as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second

form does not.

**EXAMPLE 2** To pass one function to another, one might say

```c
int f(void);
/* ... */
g(f);
```

Then the definition of \( g \) might read

```c
void g(int (*funcp)(void))
{
    /* ... */
    (*funcp)(); /* or funcp(); */
}
```

or, equivalently,

```c
void g(int func(void))
{
    /* ... */
    func(); /* or (*func)(); */
}
```

### 6.9.2 External object definitions

**Semantics**

1. If the declaration of an identifier for an object has file scope and an initializer, the declaration is an
   external definition for the identifier.

2. A declaration of an identifier for an object that has file scope without an initializer, and without a
   storage-class specifier or with the storage-class specifier `static`, constitutes a tentative definition. If a
   translation unit contains one or more tentative definitions for an identifier, and the translation unit
   contains no external definition for that identifier, then the behavior is exactly as if the translation
   unit contains a file scope declaration of that identifier, with the composite type as of the end of the
   translation unit, with an initializer equal to \( \{0\} \).

3. If the declaration of an identifier for an object is a tentative definition and has internal linkage, the
   declared type shall not be an incomplete type.
EXAMPLE 1

```c
int i1 = 1; // definition, external linkage
static int i2 = 2; // definition, internal linkage
extern int i3 = 3; // definition, external linkage
int i4; // tentative definition, external linkage
static int i5; // tentative definition, internal linkage

int i1; // valid tentative definition, refers to previous
int i2; // 6.2.2 renders undefined, linkage disagreement
int i3; // valid tentative definition, refers to previous
int i4; // valid tentative definition, refers to previous
int i5; // 6.2.2 renders undefined, linkage disagreement

extern int i1; // refers to previous, whose linkage is external
extern int i2; // refers to previous, whose linkage is internal
extern int i3; // refers to previous, whose linkage is external
extern int i4; // refers to previous, whose linkage is external
extern int i5; // refers to previous, whose linkage is internal
```

EXAMPLE 2

If at the end of the translation unit containing

```c
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.
6.10 Preprocessing directives

Syntax

preprocessing-file:
group_opt

group:
group-part

group group-part

group-part:
if-section

control-line
text-line

# # non-directive

if-section:
if-group elif-groups opt

else-group opt

endif-line

if-group:

# if constant-expression new-line group_opt

#ifdef # ifdef identifier new-line group_opt

#ifndef # ifndef identifier new-line group_opt

elif-groups:

elif-group

elif-groups elif-group

elif-group:

# elif constant-expression new-line group_opt

else-group:

# else new-line group_opt

endif-line:

# endif new-line

control-line:

#include # include pp-tokens new-line

#define # define identifier replacement-list new-line

#define # define identifier lparen identifier-list_opt ) replacement-list new-line

#define # define identifier lparen . . . ) replacement-list new-line

#define # define identifier lparen identifier-list ) replacement-list new-line

#undef # undef identifier new-line

#line # line pp-tokens new-line

#error # error pp-tokens opt new-line

#pragma # pragma pp-tokens opt new-line

# # new-line

text-line:

pp-tokens_opt new-line

non-directive:

pp-tokens new-line

lparen:
a ( (character not immediately preceded by white space

replacement-list:

pp-tokens_opt

pp-tokens:

preprocessing-token

pp-tokens preprocessing-token

new-line:

the new-line character

identifier-list:

identifier

identifier-list , identifier

§ 6.10 Language 135
Description

A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence.\(^\text{175}\) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

Semantics

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

EXAMPLE

In:

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is not a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro EMPTY has been replaced.

The execution of a non-directive preprocessing directive results in undefined behavior.

6.10.1 Conditional inclusion

Constraints

The expression that controls conditional inclusion shall be an integer constant expression except that: identifiers (including those lexically identical to keywords) are interpreted as described below;\(^\text{176}\) and it may contain unary operator expressions of the form

```
defined identifier
```

or

```
defined (identifier) (identifier)
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a #define preprocessing directive without an intervening #undef directive with the same subject identifier), 0 if it is not.

Each preprocessing token that remains (in the list of preprocessing tokens that will become the controlling expression) after all macro replacements have occurred shall be in the lexical form of a

\(^{175}\)Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 6.10.3.2, for example).

\(^{176}\)Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.
### Semantics

3 Preprocessing directives of the forms

```
#define if constant-expression new-line group_opt
#define elif constant-expression new-line group_opt
```

check whether the controlling constant expression evaluates to nonzero.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number 0, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types `intmax_t` and `uintmax_t` defined in the header `<stdint.h>`.\(^{177}\) This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.\(^{178}\) Also, whether a single-character character constant may have a negative value is implementation-defined.

5 Preprocessing directives of the forms

```
#define ifdef identifier new-line group_opt
#define ifndef identifier new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined identifier` and `#if !defined identifier` respectively.

6 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives’ preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed; any following groups are skipped and their controlling directives are processed as if they were in a group that is skipped. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.\(^{179}\)

Forward references: macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.20.1.5).

---

\(^{177}\)Thus, on an implementation where `INT_MAX` is `0x7FFF` and `UINT_MAX` is `0xFFFF`, the constant `0x8000` is signed and positive within a `#if` expression even though it would be unsigned in translation phase 7.

\(^{178}\)Thus, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in these two contexts:

```
#define if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

\(^{179}\)As indicated by the syntax, no preprocessing tokens are allowed to follow a `#else` or `#endif` directive before the terminating new-line character. However, comments can appear anywhere in a source file, including within a preprocessing directive.
6.10.2 Source file inclusion

Constraints
1 A `#include` directive shall identify a header or source file that can be processed by the implementation.

Semantics
2 A preprocessing directive of the form

```
#include <h-char-sequence>
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

```
#include "q-char-sequence"
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
#include <h-char-sequence>
```

with the identical contained sequence (including `>` characters, if any) from the original directive.

4 A preprocessing directive of the form

```
# include pp-tokens
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms. The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (6.4.2.1) followed by a period (`.`) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

7 **EXAMPLE 1** The most common uses of `#include` preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

8 **EXAMPLE 2** This illustrates macro-replaced `#include` directives:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h" // and so on
#else
#define INCFILE "versN.h"
```

Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.
Forward references: macro replacement (6.10.3).

6.10.3 Macro replacement

Constraints
1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an object-like macro shall not be redefined by another #define preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another #define preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

3 There shall be white space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation.

5 The identifier __VA_ARGS__ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics
7 The identifier immediately following the define is called the macro name. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

8 If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

```
#define identifier replacement-list new-line
```

defines an object-like macro that causes each subsequent instance of the macro name\(^{181}\) to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

```
#define identifier lparen identifier-list opt ) replacement-list new-line
#define identifier lparen ... ) replacement-list new-line
#define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a function-like macro with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the #define preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is

\(^{181}\)Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.
terminated by the matching ) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,\(^\text{182}\) the behavior is undefined.

If there is a . . . in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the . . .).

6.10.3.1 Argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument’s preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

An identifier __VA_ARGS__ that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

6.10.3.2 The # operator

Constraints

Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument’s preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of # and ## operators is unspecified.

6.10.3.3 The ## operator

Constraints

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument’s preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is

\(^{182}\)Despite the name, a non-directive is a preprocessing directive.
For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemark preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemaker preprocessing token, and concatenation of a placemaker with a non-placemark preprocessing token results in the non-placemark preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of `##` operators is unspecified.

**EXAMPLE**

In the following fragment:

```c
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
  // char p[] = "x ## y";
```

The expansion produces, at various stages:

```c
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the `##` operator.

**6.10.3.4 Rescanning and further replacement**

1. After all parameters in the replacement list have been substituted and `#` and `##` processing has taken place, all placemaker preprocessing tokens are removed. The resulting preprocessing token sequence is then rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2. If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file’s preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3. The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

4. **EXAMPLE** There are cases where it is not clear whether a replacement is nested or not. For example, given the following macro definitions:

```c
#define f(a) a+g
#define g(a) f(a)
```

the invocation

**Placemark preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.**
f(2)(9)
could expand to either
2*f(9)
or
2*9+g

Strictly conforming programs are not permitted to depend on such unspecified behavior.

### 6.10.3.5 Scope of macro definitions

1. A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

2. A preprocessing directive of the form

```
#define identifier new-line
```

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3. **EXAMPLE 1** The simplest use of this facility is to define a “manifest constant”, as in

```
#define TABSIZE 100
int table[TABSIZE];
```

4. **EXAMPLE 2** The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a): (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5. **EXAMPLE 3** To illustrate the rules for redefinition and reexamination, the sequence

```
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define z z[0]
#define h g(\~{ })
#define m(a) a(w)
#define w 0,1
#define t(a) a
#define p() int
#define q(x) x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
(f)^m(m);
p(i) i[q()] = { q(1), r(2,3), r(4,), r(5), r(,) };
char c[2][6] = { str(hello), str() };
```
<table>
<thead>
<tr>
<th>f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1);</th>
<th>f(2 * (2+(3,4)*0,1))</th>
<th>f(2 * (~{ } 5)) &amp; f(2 * (0,1))^m(0,1);</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i[] = { 1, 23, 4, 5, };</td>
<td>char c[2][6] = { &quot;hello&quot;, &quot;&quot; };</td>
<td></td>
</tr>
</tbody>
</table>
EXAMPLE 4  To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s) # s
#define xstr(s) str(s)
#define debug(s, t) printf("x" # s " = %d, x" # t " = %s", \
      x ## s, x ## t)
#define INCFILE(n) vers ## n
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW "hello"
#define LOW LOW ", world"

debug(1, 2);
fputs(str(strlen("abc\0d", "abc", '\4') // this goes away
      == 0) str(: @
), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strlen("abc\0d", "abc", '\4') == 0: @\n", s);
#include "vers2.h"  (after macro replacement, before file access)
"hello";
"hello", "world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strlen("abc\0d", "abc", '\4') == 0: @\n", s);
#include "vers2.h"  (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

EXAMPLE 5  To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,), 
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89, 
           10, 11, 12,  };
```

EXAMPLE 6  To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) (a)
#define FUNC_LIKE(a) /* note the white space */ \
                 a /* other stuff on this line */
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) (a)
#define FUNC_LIKE(a) /* note the white space */ \
                 a /* other stuff on this line */
```
A preprocessing directive of the form

```c
#define OBJLIKE (0)  // different token sequence
#define OBJLIKE (1 - 1) // different white space
#define FUNCLIKE(b) (a) // different parameter usage
#define FUNCLIKE(b) (b) // different parameter spelling
```

EXAMPLE 7 Finally, to show the variable argument list macro facilities:

```c
#define debug(...)  fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(__VA_ARGS__)
#define report(test, ...) {((test)?puts(#test):
    printf(__VA_ARGS__))

debug("Flag");
d debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```c
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts(The first, second, and third items.);
((x>y)?puts("x>y"):
    printf("x is %d but y is %d", x, y));
```

6.10.4 Line control

Constraints
1. The string literal of a `#line` directive, if present, shall be a character string literal.

Semantics
2. The line number of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.2) while processing the source file to the current token.

3. If a preprocessing token (in particular `__LINE__`) spans two or more physical lines, it is unspecified which of those line numbers is associated with that token. If a preprocessing directive spans two or more physical lines, it is unspecified which of those line numbers is associated with the preprocessing directive. If a macro invocation spans multiple physical or logical lines, it is unspecified which of those line numbers is associated with that invocation. The line number of a preprocessing token is independent of the context (in particular, as a macro argument or in a preprocessing directive). The line number of a `__LINE__` in a macro body is the line number of the macro invocation.

4. A preprocessing directive of the form

```c
  # line digit-sequence new-line
```

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647.

5. A preprocessing directive of the form

```c
  # line digit-sequence " s-char-sequence_opt " new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

6. A preprocessing directive of the form

```c
  # line pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after
all replacements shall match one of the two previous forms and is then processed as appropriate.\textsuperscript{184)}

\textbf{Recommended practice}

7 The line number associated with a \texttt{pp-token} should be the line number of the first character of the \texttt{pp-token}. The line number associated with a preprocessing directive should be the line number of the line with the first \# token. The line number associated with a macro invocation should be the line number of the first character of the macro name in the invocation.

\section*{6.10.5 Error directive}

\textbf{Semantics}

1 A preprocessing directive of the form

\texttt{\# \# error \texttt{pp-tokens}\textsubscript{opt} new-line}

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

\section*{6.10.6 Pragma directive}

\textbf{Semantics}

1 A preprocessing directive of the form

\texttt{\# \# \texttt{pragma} \texttt{pp-tokens}\textsubscript{opt} new-line}

where the preprocessing token \texttt{STDC-STDC} does not immediately follow \texttt{pragma} in the directive (prior to any macro replacement)\textsuperscript{185)} causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such \texttt{pragma} that is not recognized by the implementation is ignored.

2 If the preprocessing token \texttt{STDC} does immediately follow \texttt{pragma} in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms\textsuperscript{186)} whose meanings are described elsewhere:

\begin{verbatim}
# standard-pragmas:
# pragma STDC_FP_CONTRACT on-off-switch
# pragma STDC_FENV_ACCESS on-off-switch
# pragma STDC_FENV_DEC_ROUND dec-direction
# pragma STDC_FENV_ROUND direction
# pragma STDC_CX_LIMITED_RANGE on-off-switch
\end{verbatim}

\begin{description}
\item[on-off-switch: one of] ON OFF DEFAULT ON OFF DEFAULT
\item[direction: one of] FE_DOWNWARD FE_TONEAREST FE_TONEARESTFROMZERO
FE_TOWARDZERO FE_UPWARD FE_DYNAMIC
\item[dec-direction: one of] FE_DEC_DOWNWARD FE_DEC_TONEAREST FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO FE_DEC_UPWARD FE_DEC_DYNAMIC
\end{description}

\textsuperscript{184)}Because a new-line is explicitly included as part of the \texttt{#line} directive, the number of new-line characters read while processing to the first \texttt{pp-token} can be different depending on whether or not the implementation uses a one-pass preprocessor. Therefore, there are two possible values for the line number following a directive of the form \texttt{#line \_\_\_LINE\_ new-line}.

\textsuperscript{185)}An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where \texttt{STDC} immediately follows \texttt{pragma}). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

\textsuperscript{186)}See “future language directions” (6.11.7).
Forward references: the `FP_CONTRACT` pragma (7.12.2), the `FENV_ACCESS` pragma (7.6.1), the `FENV_DEC_ROUND` pragma (7.6.3), the `FENV_ROUND` pragma (7.6.2), the `CX_LIMITED_RANGE` pragma (7.3.4).

6.10.7 Null directive

Semantics

1. A preprocessing directive of the form

```
    # new-line
```

has no effect.

6.10.8 Predefined macro names

1. The values of the predefined macros listed in the following subclauses\(^\text{187}\) (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.

2. None of these macro names, nor the identifier `defined`, shall be the subject of a `#define` or a `#undef` preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

3. The implementation shall not redefine the macro `__cplusplus`, nor shall it define it in any standard header.

Forward references: standard headers (7.1.2).

6.10.8.1 Mandatory macros

1. The following macro names shall be defined by the implementation:

\(^{187}\)See “future language directions” (6.11.8).
__DATE__ — The date of translation of the preprocessing translation unit: a character string literal of the form "Mmm dd yyyy", where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

__FILE__ — The presumed name of the current source file (a character string literal).\(^{188}\)

__LINE__ — The presumed line number (within the current source file) of the current source line (an integer constant).\(^{188}\)

__STDC__ — The integer constant 1, intended to indicate a conforming implementation.

__STDC_HOSTED__ — The integer constant 1 if the implementation is a hosted implementation or the integer constant 0 if it is not.

__STDC_VERSION__ — The integer constant `yyyymmL`.\(^{189}\)

__TIME__ — The time of translation of the preprocessing translation unit: a character string literal of the form "hh:mm:ss" as in the time generated by the `asctime` function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

Forward references: the `asctime` function (7.27.3.1).

6.10.8.2 Environment macros

The following macro names are conditionally defined by the implementation:

__STDC_ISO_10646__ — An integer constant of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

__STDC_MB_MIGHT_NEQ_WC__ — The integer constant 1, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant.

__STDC_UTF_16__ — The integer constant 1, intended to indicate that values of type `char16_t` are UTF–16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

__STDC_UTF_32__ — The integer constant 1, intended to indicate that values of type `char32_t` are UTF–32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

Forward references: common definitions (7.19), unicode utilities (7.28).

6.10.8.3 Conditional feature macros

The following macro names are conditionally defined by the implementation:

__STDC_ANALYZABLE__ — The integer constant 1, intended to indicate conformance to the specifications in Annex L (Analyzability).

\(^{188}\) The presumed source file name and line number can be changed by the `#line` directive.

\(^{189}\) See Annex M for the values in previous revisions. The intention is that this will remain an integer constant of type `long int` that is increased with each revision of this document.

Language § 6.10.8.3
__STDC_IEC_60559_BFP__ The integer constant yyyymm, intended to indicate conformance to Annex F (IEC 60559 binary floating-point arithmetic).

__STDC_IEC_559__ The integer constant 1, intended to indicate conformance to the specifications in Annex F (IEC 60559 floating-point arithmetic). Use of this macro is an obsolescent feature.

__STDC_IEC_60559_DFP__ The integer constant yyyymmL, intended to indicate support of decimal floating types and conformance to Annex F for IEC 60559 decimal floating-point arithmetic.

__STDC_IEC_60559_COMPLEX__ The integer constant yyyymmL, intended to indicate conformance to the specifications in Annex G (IEC 60559 compatible complex arithmetic).

__STDC_IEC_559_COMPLEX__ The integer constant 1, intended to indicate adherence to the specifications in Annex G (IEC 60559 compatible complex arithmetic). Use of this macro is an obsolescent feature.

__STDC_LIB_EXT1__ The integer constant yyyymm, intended to indicate support for the extensions defined in Annex K (Bounds-checking interfaces).

__STDC_NO_ATOMICS__ The integer constant 1, intended to indicate that the implementation does not support atomic types (including the _Atomic type qualifier) and the `<stdatomic.h>` header.

__STDC_NO_COMPLEX__ The integer constant 1, intended to indicate that the implementation does not support complex types or the `<complex.h>` header.

__STDC_NO_THREADS__ The integer constant 1, intended to indicate that the implementation does not support the `<threads.h>` header.

__STDC_NO_VLA__ The integer constant 1, intended to indicate that the implementation does not support variable length arrays or variably modified types.

An implementation that defines __STDC_NO_COMPLEX__ shall not define __STDC_IEC_60559_COMPLEX__ or __STDC_IEC_559_COMPLEX__.

6.10.9 Pragma operator

Semantics

A unary operator expression of the form:

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is destringized by deleting any encoding prefix, deleting the leading and trailing double-quotes, replacing each escape sequence " by a double-quote, and replacing each escape sequence \ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the pp-tokens in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

EXAMPLE A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ("listing on \"..\listing.dir\"")
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

190) The intention is that this will remain an integer constant of type long int that is increased with each revision of this document.
```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING (..\listing.dir)
```
6.11 Future language directions

6.11.1 Floating types
1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than long double.

6.11.2 Linkages of identifiers
1 Declaring an identifier with internal linkage at file scope without the static storage-class specifier is an obsolescent feature.

6.11.3 External names
1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences
1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Storage-class specifiers
1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.6 Function declarators
1 The use of function declarators with empty parentheses (not prototype format parameter type declarators) without prototypes is an obsolescent feature. The use of function definitions with separate parameter identifier and declaration lists (not prototype format parameter type and identifier declarators) is an obsolescent feature.

6.11.7Pragma directives
1 Pragmas whose first preprocessing token is STDC are reserved for future standardization.

6.11.8 Predefined macro names
1 Macro names beginning with __STDC__ are reserved for future standardization.

2 Uses of the __STDC_IEC_559__ and __STDC_IEC_559_COMPLEX__ macros are obsolescent features.
7. Library

7.1 Introduction

7.1.1 Definitions of terms

1 A string is a contiguous sequence of characters terminated by and including the first null character. The term multibyte string is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A pointer to a string is a pointer to its initial (lowest addressed) character. The length of a string is the number of bytes preceding the null character and the value of a string is the sequence of the values of the contained characters, in order.

2 The decimal-point character is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.\(^{191}\) It is represented in the text and examples by a period, but may be changed by the setlocale function.

3 A null wide character is a wide character with code value zero.

4 A wide string is a contiguous sequence of wide characters terminated by and including the first null wide character. A pointer to a wide string is a pointer to its initial (lowest addressed) wide character. The length of a wide string is the number of wide characters preceding the null wide character and the value of a wide string is the sequence of code values of the contained wide characters, in order.

5 A shift sequence is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state (see 5.2.1.2). A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.\(^{192}\) In this clause, references to “white-space character” refer to (execution) white-space character as defined by isspace. References to “white-space wide character” refer to (execution) white-space wide character as defined by iswspace.

Forward references: character handling (7.4), the setlocale function (7.11.1.1).

7.1.2 Standard headers

1 Each library function is declared, with a type that includes a prototype, in a header,\(^{193}\) whose contents are made available by the include preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use. In addition to the provisions given in this clause, an implementation that defines __STDC_LIB_EXT1__ shall conform to the specifications in Annex K and Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of this clause. Declarations of types described here or in Annex K shall not include type qualifiers, unless explicitly stated otherwise.

2 An implementation that does not support decimal floating types (6.10.8.3) need not support interfaces or aspects of interfaces that are specific to these types.

3 The standard headers are\(^{194}\)

\[
\begin{align*}
\text{<assert.h>}& \quad \text{<errno.h>}& \quad \text{<inttypes.h>}
\text{<complex.h>}& \quad \text{<fenv.h>}& \quad \text{<iso646.h>}
\text{<ctype.h>}& \quad \text{<float.h>}& \quad \text{<limits.h>}
\end{align*}
\]

\(^{191}\)The functions that make use of the decimal-point character are the numeric conversion functions (7.22.1, 7.29.1) and the formatted input/output functions (7.21.6, 7.29.2).

\(^{192}\)For state-dependent encodings, the values for MB_CUR_MAX and MB_LEN_MAX are thus required to be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation’s choice.

\(^{193}\)A header is not necessarily a source file, nor are the < and > delimited sequences in header names necessarily valid source file names.

\(^{194}\)The headers <complex.h>, <stdatomic.h>, and <threads.h> are conditional features that implementations need not support; see 6.10.8.3.
Each header declares or defines all identifiers listed in its associated subclause, and optionally declares identifiers listed in its associated subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

1. All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use, except those identifiers which are lexically identical to keywords.\(^1\)

2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.

3. Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 7.1.4).

4. All identifiers with external linkage in any of the following subclauses (including the future library directions) and \texttt{errno} are always reserved for use as identifiers with external linkage.\(^2\)

5. Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

\(^1\) Allows identifiers spelled with a leading underscore followed by an uppercase letter that match the spelling of a keyword to be used as macro names by the program.

\(^2\) The list of reserved identifiers with external linkage includes \texttt{math_errhandling}, \texttt{setjmp}, \texttt{va_copy}, and \texttt{va_end}.
2 No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 7.1.4), or defines a reserved identifier or attribute token described in 6.7.11.1 as a macro name, the behavior is undefined.

3 If the program removes (with `#undef`) any macro definition of an identifier in the first group listed above or attribute token described in 6.7.11.1, the behavior is undefined.

### 7.1.4 Use of library functions

1 Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow:

- If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after default argument promotion) not expected by a function with a variable number of arguments, the behavior is undefined.

- If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

- Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.\(^{197}\)

   The use of `#undef` to remove any macro definition will also ensure that an actual function is referred to.

- Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.\(^{198}\)

- Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.\(^{199}\)

- All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in `#if` preprocessing directives.

2 Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

3 There is a sequence point immediately before a library function returns.

---

\(^{197}\)This means that an implementation is required to provide an actual function for each library function, even if it also provides a macro for that function.

\(^{198}\)Such macros might not contain the sequence points that the corresponding function calls do.

\(^{199}\)Because external identifiers and some macro names beginning with an underscore are reserved, implementations can provide special semantics for such names. For example, the identifier `_BUILTIN_abs` could be used to indicate generation of in-line code for the `abs` function. Thus, the appropriate header could specify

```c
#define abs(x) _BUILTIN_abs(x)
```

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as `abs` will be a genuine function can write

```c
#define abs(x) _BUILTIN_abs(x)
```

whether the implementation’s header provides a macro implementation of `abs` or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.
The functions in the standard library are not guaranteed to be reentrant and may modify objects with static or thread storage duration.

Unless explicitly stated otherwise in the detailed descriptions that follow, library functions shall prevent data races as follows: A library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s arguments. A library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function’s non-const arguments. Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

Unless otherwise specified, library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.

EXAMPLE The function `atoi` can be used in any of several ways:

— by use of its associated header (possibly generating a macro expansion)

```c
#include <stdlib.h>
const char *str;
/* ... */
i = atoi(str);
```

— by use of its associated header (assuredly generating a true function reference)

```c
#include <stdlib.h>
#undef atoi
const char *str;
/* ... */
i = atoi(str);
```

or

```c
#include <stdlib.h>
const char *str;
/* ... */
i = (atoi)(str);
```

— by explicit declaration

```c
extern int atoi(const char *);
const char *str;
/* ... */
i = atoi(str);
```

Thus, a signal handler cannot, in general, call standard library functions.

This means, for example, that an implementation is not permitted to use a `static` object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. Similarly, an implementation of `memcpy` is not permitted to copy bytes beyond the specified length of the destination object and then restore the original values because it could cause a data race if the program shared those bytes between threads.

This allows implementations to parallelize operations if there are no visible side effects.
7.2 Diagnostics <assert.h>

The header <assert.h> defines the `assert` and `static_assert` macros and refers to another macro,

```c
#define assert(ignore) ((void)0)
```

which is not defined by <assert.h>. If `NDEBUG` is defined as a macro name at the point in the source file where <assert.h> is included, the `assert` macro is defined simply as

The `assert` macro is redefined according to the current state of `NDEBUG` each time that <assert.h> is included.

The `assert` macro shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

The macro

```c
static_assert
```

expands to `__Static_assert`.

### 7.2.1 Program diagnostics

#### 7.2.1.1 The assert macro

**Synopsis**

```c
#include <assert.h>
void assert(scalar expression);
```

**Description**

The `assert` macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if `expression` (which shall have a scalar type) is false (that is, compares equal to 0), the `assert` macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function — the latter are respectively the values of the preprocessing macros `__FILE__` and `__LINE__` and of the identifier `__func__`) on the standard error stream in an implementation-defined format.  It then calls the `abort` function.

**Returns**

The `assert` macro returns no value.

**Forward references:** the `abort` function (7.22.4.1).

---

203) The message written might be of the form:

```
Assertion failed: expression, function abc, file xyz, line nnn.
```
7.3 Complex arithmetic <complex.h>

7.3.1 Introduction

The header <complex.h> defines macros and declares functions that support complex arithmetic.

Implementations that define the macro __STDC_NO_COMPLEX__ need not provide this header nor support any of its facilities.

Each synopsis, other than for the CMPLX macros, specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex or double return value; and other functions with the same name but with f and l suffixes which are corresponding functions with float and long double parameters and return values.

The macro

```c
complex
```
expands to _Complex; the macro

```c
__Complex_I
```
expands to a constant expression of type const float _Complex, with the value of the imaginary unit.

The macros

```c
imaginary
```
and

```c
__Imaginary_I
```
are defined if and only if the implementation supports imaginary types; if defined, they expand to _Imaginary and a constant expression of type const float _Imaginary with the value of the imaginary unit.

The macro

```c
I
```
expands to either _Imaginary_I or _Complex_I. If _Imaginary_I is not defined, I shall expand to _Complex_I.

Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros complex, imaginary, and I.

Forward references: the CMPLX macros (7.3.9.3), IEC 60559-compatible complex arithmetic (Annex G).

7.3.2 Conventions

Values are interpreted as radians, not degrees. An implementation may set errno but is not required to.

7.3.3 Branch cuts

Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications of Annex G, the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For

---

204) See “future library directions” (7.31.1).  
205) The imaginary unit is a number \( i \) such that \( i^2 = -1 \).  
206) A specification for imaginary types is in Annex G.
example, for the square root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part -0, maps to the negative imaginary axis.

Implementations that do not support a signed zero (see Annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

### 7.3.4 The CX_LIMITED_RANGE pragma

**Synopsis**

```c
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

**Description**

The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The `CX_LIMITED_RANGE` pragma can be used to inform the implementation that (where the state is “on”) the usual mathematical formulas are acceptable. The pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is “off”.

### 7.3.5 Trigonometric functions

#### 7.3.5.1 The cacos functions

**Synopsis**

```c
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

**Description**

The `cacos` functions compute the complex arc cosine of `z`, with branch cuts outside the interval \([-1, +1]\) along the real axis.

**Returns**

The `cacos` functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval \([0, \pi]\) along the real axis.

---

207) The purpose of the pragma is to allow the implementation to use the formulas:

\[
(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv) \\
(x + iy) / (u + iv) = \frac{(xu + yv) + i(yu - xv)}{(u^2 + v^2)} \\
|x + iy| = \sqrt{x^2 + y^2}
\]

where the programmer can determine they are safe.
7.3.5.2 The `casin` functions

**Synopsis**

```c
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

**Description**

The `casin` functions compute the complex arc sine of `z`, with branch cuts outside the interval `[-1, +1]` along the real axis.

**Returns**

The `casin` functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval `[−π/2, +π/2]` along the real axis.

7.3.5.3 The `catan` functions

**Synopsis**

```c
#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

**Description**

The `catan` functions compute the complex arc tangent of `z`, with branch cuts outside the interval `[-i, +i]` along the imaginary axis.

**Returns**

The `catan` functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval `[−π/2, +π/2]` along the real axis.

7.3.5.4 The `ccos` functions

**Synopsis**

```c
#include <complex.h>

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

**Description**

The `ccos` functions compute the complex cosine of `z`.

**Returns**

The `ccos` functions return the complex cosine value.

7.3.5.5 The `csin` functions

**Synopsis**

```c
#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

**Description**

The `csin` functions compute the complex sine of `z`.

**Returns**

The `csin` functions return the complex sine value.
7.3.5.6 The \texttt{ctan} functions

**Synopsis**

```c
#include <complex.h>

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

**Description**

The \texttt{ctan} functions compute the complex tangent of \( z \).

**Returns**

The \texttt{ctan} functions return the complex tangent value.

7.3.6 Hyperbolic functions

7.3.6.1 The \texttt{cacosh} functions

**Synopsis**

```c
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

**Description**

The \texttt{cacosh} functions compute the complex arc hyperbolic cosine of \( z \), with a branch cut at values less than 1 along the real axis.

**Returns**

The \texttt{cacosh} functions return the complex arc hyperbolic cosine value, in the range of a half-strip of nonnegative values along the real axis and in the interval \([-i\pi, +i\pi]\) along the imaginary axis.

7.3.6.2 The \texttt{casinh} functions

**Synopsis**

```c
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

**Description**

The \texttt{casinh} functions compute the complex arc hyperbolic sine of \( z \), with branch cuts outside the interval \([-i, +i]\) along the imaginary axis.

**Returns**

The \texttt{casinh} functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval \([-i\pi/2, +i\pi/2]\) along the imaginary axis and in the interval \([-i\pi/2, +i\pi/2]\) along the imaginary axis.

7.3.6.3 The \texttt{catanh} functions

**Synopsis**

```c
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

**Description**

The \texttt{catanh} functions compute the complex arc hyperbolic tangent of \( z \), with branch cuts outside the interval \([-1, +1]\) along the real axis.
Returns

3 The \texttt{catanh} functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval \([-\pi/2, +\pi/2] \cup \left(-\frac{\pi}{2}, +\frac{\pi}{2}\right]\) along the imaginary axis.

7.3.6.4 The \texttt{ccosh} functions

Synopsis

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Description

2 The \texttt{ccosh} functions compute the complex hyperbolic cosine of \(z\).

Returns

3 The \texttt{ccosh} functions return the complex hyperbolic cosine value.

7.3.6.5 The \texttt{csinh} functions

Synopsis

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Description

2 The \texttt{csinh} functions compute the complex hyperbolic sine of \(z\).

Returns

3 The \texttt{csinh} functions return the complex hyperbolic sine value.

7.3.6.6 The \texttt{ctanh} functions

Synopsis

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Description

2 The \texttt{ctanh} functions compute the complex hyperbolic tangent of \(z\).

Returns

3 The \texttt{ctanh} functions return the complex hyperbolic tangent value.

7.3.7 Exponential and logarithmic functions

7.3.7.1 The \texttt{cexp} functions

Synopsis

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Description

2 The \texttt{cexp} functions compute the complex base-\(e\) exponential of \(z\).
Returns
3 The \texttt{cexp} functions return the complex base-$e$ exponential value.

7.3.7.2 The \texttt{clog} functions

Synopsis
1
\begin{verbatim}
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
\end{verbatim}

Description
2 The \texttt{clog} functions compute the complex natural (base-$e$) logarithm of \(z\), with a branch cut along the negative real axis.

Returns
3 The \texttt{clog} functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval \([-i\pi, +i\pi]\) along the imaginary axis.

7.3.8 Power and absolute-value functions

7.3.8.1 The \texttt{cabs} functions

Synopsis
1
\begin{verbatim}
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
\end{verbatim}

Description
2 The \texttt{cabs} functions compute the complex absolute value (also called norm, modulus, or magnitude) of \(z\).

Returns
3 The \texttt{cabs} functions return the complex absolute value.

7.3.8.2 The \texttt{cpow} functions

Synopsis
1
\begin{verbatim}
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
\end{verbatim}

Description
2 The \texttt{cpow} functions compute the complex power function \(x^y\), with a branch cut for the first parameter along the negative real axis.

Returns
3 The \texttt{cpow} functions return the complex power function value.
7.3.8.3 The csqrt functions

Synopsis

```c
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Description

The `csqrt` functions compute the complex square root of `z`, with a branch cut along the negative real axis.

Returns

The `csqrt` functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

7.3.9 Manipulation functions

7.3.9.1 The carg functions

Synopsis

```c
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Description

The `carg` functions compute the argument (also called phase angle) of `z`, with a branch cut along the negative real axis.

Returns

The `carg` functions return the value of the argument in the interval \([-\pi, +\pi]\).

7.3.9.2 The cimag functions

Synopsis

```c
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

Description

The `cimag` functions compute the imaginary part of `z`.

Returns

The `cimag` functions return the imaginary part value (as a real).

7.3.9.3 The CMPLX macros

Synopsis

```c
#include <complex.h>
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
```

Description

The `CMPLX` macros expand to an expression of the specified complex type, with the real part having the (converted) value of `x` and the imaginary part having the (converted) value of `y`. The resulting

\[ z = \text{creal}(z) + \text{cimag}(z) \times \mathbf{i} . \]
expression shall be suitable for use as an initializer for an object with static or thread storage duration, provided both arguments are likewise suitable.

Returns

3 The CMPLX macros return the complex value \( x + iy \).

4 NOTE These macros act as if the implementation supported imaginary types and the definitions were:

```c
#define CMPLX(x, y) ((double complex)((double)(x) + _Imaginary_I * (double)(y)))
#define CMPLXF(x, y) ((float complex)((float)(x) + _Imaginary_I * (float)(y)))
#define CMPLXL(x, y) ((long double complex)((long double)(x) + _Imaginary_I * (long double)(y)))
```

7.3.9.4 The `conj` functions

Synopsis

```c
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Description

2 The `conj` functions compute the complex conjugate of \( z \), by reversing the sign of its imaginary part.

Returns

3 The `conj` functions return the complex conjugate value.

7.3.9.5 The `cproj` functions

Synopsis

```c
#include <complex.h>

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Description

2 The `cproj` functions compute a projection of \( z \) onto the Riemann sphere: \( z \) projects to \( z \) except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If \( z \) has an infinite part, then `cproj(z)` is equivalent to

```
INFINITY + I * copysign(0.0, cimag(z))
```

Returns

3 The `cproj` functions return the value of the projection onto the Riemann sphere.

7.3.9.6 The `creal` functions

Synopsis

```c
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Description

2 The `creal` functions compute the real part of \( z \).

### Footnote

For a variable \( z \) of complex type, \( z == \text{creal}(z) + \text{cimag}(z) \times I \).
Returns
3 The **creal** functions return the real part value.
7.4 Character handling `<ctype.h>`

The header `<ctype.h>` declares several functions useful for classifying and mapping characters.\(^{210}\) In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall equal the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the "C" locale are noted below.

The term *printing character* refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a locale-specific set of characters that are not printing characters.\(^{211}\) All letters and digits are printing characters.

**Forward references:** `EOF` (7.21.1), localization (7.11).

### 7.4.1 Character classification functions

The functions in this subclause return nonzero (true) if and only if the value of the argument `c` conforms to that in the description of the function.

#### 7.4.1.1 The `isalnum` function

**Synopsis**

```c
#include <ctype.h>
int isalnum(int c);
```

**Description**

The `isalnum` function tests for any character for which `isalpha` or `isdigit` is true.

#### 7.4.1.2 The `isalpha` function

**Synopsis**

```c
#include <ctype.h>
int isalpha(int c);
```

**Description**

The `isalpha` function tests for any character for which `isupper` or `islower` is true, or any character that is one of a locale-specific set of alphabetic characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true.\(^{212}\) In the "C" locale, `isalpha` returns true only for the characters for which `isupper` or `islower` is true.

#### 7.4.1.3 The `isblank` function

**Synopsis**

```c
#include <ctype.h>
int isblank(int c);
```

**Description**

The `isblank` function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which `isspace` is true and that is used to separate words within a line of text. The standard blank characters are the following: space (’ ’), and horizontal tab (’\t ’). In the "C" locale, `isblank` returns true only for the standard blank characters.

\(^{210}\) See “future library directions” (7.31.2).

\(^{211}\) In an implementation that uses the seven-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde); the control characters are those whose values lie from 0 (NULL) through 0x1F (US), and the character 0x7F (DEL).

\(^{212}\) The functions `islower` and `isupper` test true or false separately for each of these additional characters; all four combinations are possible.
7.4.1.4 The iscntrl function

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Description

The `iscntrl` function tests for any control character.

7.4.1.5 The isdigit function

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Description

The `isdigit` function tests for any decimal-digit character (as defined in 5.2.1).

7.4.1.6 The isgraph function

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Description

The `isgraph` function tests for any printing character except space (’ ’).

7.4.1.7 The islower function

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

The `islower` function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the "C" locale, `islower` returns true only for the lowercase letters (as defined in 5.2.1).

7.4.1.8 The isprint function

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Description

The `isprint` function tests for any printing character including space (’ ’).

7.4.1.9 The ispunct function

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

The `ispunct` function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither `isspace` nor `isalnum` is true. In the "C" locale, `ispunct` returns true for every printing character for which neither `isspace` nor `isalnum` is true.

§ 7.4.9 Library
7.4.1.10 The **isspace** function

**Synopsis**

```c
#include <cctype>
int isspace(int c);
```

**Description**

The **isspace** function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which **isalnum** is false. The standard white-space characters are the following: space (’ ’), form feed (’\f’), new-line (’\n’), carriage return (’\r’), horizontal tab (’\t’), and vertical tab (’\v’). In the “C” locale, **isspace** returns true only for the standard white-space characters.

7.4.1.11 The **isupper** function

**Synopsis**

```c
#include <cctype>
int isupper(int c);
```

**Description**

The **isupper** function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the “C” locale, **isupper** returns true only for the uppercase letters (as defined in 5.2.1).

7.4.1.12 The **isxdigit** function

**Synopsis**

```c
#include <cctype>
int isxdigit(int c);
```

**Description**

The **isxdigit** function tests for any hexadecimal-digit character (as defined in 6.4.4.1).

7.4.2 Character case mapping functions

7.4.2.1 The **tolower** function

**Synopsis**

```c
#include <cctype>
int tolower(int c);
```

**Description**

The **tolower** function converts an uppercase letter to a corresponding lowercase letter.

**Returns**

If the argument is a character for which **isupper** is true and there are one or more corresponding characters, as specified by the current locale, for which **islower** is true, the **tolower** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

7.4.2.2 The **toupper** function

**Synopsis**

```c
#include <cctype>
int toupper(int c);
```

**Description**

The **toupper** function converts a lowercase letter to a corresponding uppercase letter.
Returns

If the argument is a character for which `islower` is true and there are one or more corresponding characters, as specified by the current locale, for which `isupper` is true, the `toupper` function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.
7.5 Errors <errno.h>

The header <errno.h> defines several macros, all relating to the reporting of error conditions.

The macros are:

```
EDOM
EILSEQ
ERANGE
```

which expand to integer constant expressions with type `int`, distinct positive values, and which are suitable for use in `#if` preprocessing directives; and

```
errno
```

which expands to a modifiable lvalue\(^\text{213}\) that has type `int` and thread local storage duration, the value of which is set to a positive error number by several library functions. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name `errno`, the behavior is undefined.

The value of `errno` in the initial thread is zero at program startup (the initial value of `errno` in other threads is an indeterminate value), but is never set to zero by any library function.\(^\text{214}\) The value of `errno` may be set to nonzero by a library function call whether or not there is an error, provided the use of `errno` is not documented in the description of the function in this document.

Additional macro definitions, beginning with `E` and a digit or `E` and an uppercase letter,\(^\text{215}\) may also be specified by the implementation.

---

\(^{213}\)The macro `errno` need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, `errno()`).

\(^{214}\)Thus, a program that uses `errno` for error checking would set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of `errno` on entry and then set it to zero, as long as the original value is restored if `errno`'s value is still zero just before the return.

\(^{215}\)See “future library directions” (7.31.3).
7.6 Floating-point environment `<fenv.h>

The header `<fenv.h>` defines several macros, and declares types and functions that provide access to the floating-point environment. The floating-point environment refers collectively to any floating-point status flags and control modes supported by the implementation. A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information. A floating-point control mode is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

The floating-point control mode may be constant (7.6.2) or dynamic. The dynamic floating-point environment includes the dynamic floating-point control modes and the floating-point status flags.

The dynamic floating-point environment has thread storage duration. The initial state for a thread’s dynamic floating-point environment is the current state of the dynamic floating-point environment of the thread that creates it at the time of creation.

Certain programming conventions support the intended model of use for the dynamic floating-point environment:

- a function call does not alter its caller’s floating-point control modes, clear its caller’s floating-point status flags, nor depend on the state of its caller’s floating-point status flags unless the function is so documented;
- a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;
- a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

The feature test macro `__STDC_VERSION_FENV_H__` expands to the token `yyyymmL`.

The type

```
fenv_t
```

represents the entire dynamic floating-point environment.

The type

```
femode_t
```

represents the collection of dynamic floating-point control modes supported by the implementation, including the dynamic rounding direction mode.

The type

```
fexcept_t
```

represents the floating-point status flags collectively, including any status the implementation associates with the flags.

Each of the macros

```
FE_DIVBYZERO
FE_INEXACT
```

---

<sup>216</sup>This header is designed to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. It is also designed to facilitate code portability among all systems.

<sup>217</sup>A floating-point status flag is not an object and can be set more than once within an expression.

<sup>218</sup>With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.
Each of the macros are defined if and only if the implementation supports the floating-point exception by means of the functions in 7.6.4. Additional implementation-defined floating-point exceptions, with macro definitions beginning with \texttt{FE} and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values, and furthermore, bitwise ANDs of all combinations of the macros result in zero.\footnote{The implementation supports a floating-point exception if there are circumstances where a call to at least one of the functions in 7.6.4, using the macro as the appropriate argument, will succeed. It is not necessary for all the functions to succeed all the time.}

Decimal floating-point operations and IEC 60559 binary floating-point operations (Annex F) access the same floating-point exception status flags.

The macro \texttt{FE\_DFL\_MODE} represents the default state for the collection of dynamic floating-point control modes supported by the implementation and has type \texttt{“pointer to const-qualified femode_t”}. Additional implementation-defined states for the dynamic mode collection, with macro definitions beginning with \texttt{FE} and an uppercase letter, and having type \texttt{“pointer to const-qualified femode_t”}, may also be specified by the implementation.

The macro \texttt{FE\_ALL\_EXCEPT} is simply the bitwise OR of all floating-point exception macros defined by the implementation. If no such macros are defined, \texttt{FE\_ALL\_EXCEPT} shall be defined as 0.

Each of the macros is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the \texttt{fesetround} and \texttt{fegetround} functions. Additional implementation-defined rounding directions, with macro definitions beginning with \texttt{FE} and an uppercase letter, may also be specified by the implementation.\footnote{See “future library directions” (7.31.4).}

If the implementation supports decimal floating types, each of the macros:

is defined for use with the \texttt{fe\_dec\_getround} and \texttt{fe\_dec\_setround} functions for getting and setting the dynamic rounding direction mode, and with the \texttt{FENV\_DEC\_ROUND} rounding control

\footnote{The macros are typically distinct powers of two.}

\footnote{See “future library directions” (7.31.4).}

\footnote{Even though the rounding direction macros might expand to constants corresponding to the values of \texttt{FLT\_ROUNDS}, they are not required to do so.}
pragma (7.6.3) for specifying a constant rounding direction, for decimal floating-point operations. The decimal rounding direction affects all (inexact) operations that produce a result of decimal floating type and all operations that produce an integer or character sequence result and have an operand of decimal floating type, unless stated otherwise. The macros expand to integer constant expressions whose values are distinct nonnegative values.

During translation, constant rounding direction modes for decimal floating-point arithmetic are in effect where specified. Elsewhere, during translation the decimal rounding direction mode is FE_DEC_TONEAREST.

At program startup the dynamic rounding direction mode for decimal floating-point arithmetic is initialized to FE_DEC_TONEAREST.

The macro

```c
#define FE_DFL_ENV
```

represents the default dynamic floating-point environment — the one installed at program startup — and has type “pointer to const-qualified fenv_t”. It can be used as an argument to <fenv.h> functions that manage the dynamic floating-point environment.

Additional implementation-defined environments, with macro definitions beginning with FE_ and an uppercase letter,\(^{224}\) and having type “pointer to const-qualified fenv_t”, may also be specified by the implementation.

### 7.6.1 The FENV_ACCESS pragma

**Synopsis**

```c
#include <fenv.h>
#define STDC FENV_ACCESS on-off-switch
```

**Description**

The FENV_ACCESS pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.\(^{225}\) The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another FENV_ACCESS pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another FENV_ACCESS pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags, sets floating-point control modes, or runs under other establishes non-default mode settings floating-point mode settings using any means other than the FENV_ROUND pragmas, but was translated with the state for the FENV_ACCESS pragma “off”, the behavior is undefined. The default state (“on” or “off”) for the pragma is implementation-defined. (When execution passes from a part of the program translated with FENV_ACCESS “off” to a part translated with FENV_ACCESS “on”, the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

**EXAMPLE**

```c
#include <fenv.h>
void f(double x)
{
```

\(^{224}\) See “future library directions” (7.31.4).

\(^{225}\) The purpose of the FENV_ACCESS pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of FENV_ACCESS is “off”, the translator can assume that the flags are not tested, and that default modes are in effect, except where specified otherwise by an FENV_ROUND pragma.


```c
#pragma STDC FENV_ACCESS ON
void g(double);
void h(double);
/* ... */
g(x + 1);
h(x + 1);
/* ... */
```

4 If the function $g$ might depend on status flags set as a side effect of the first $x + 1$, or if the second $x + 1$ might depend on control modes set as a side effect of the call to function $g$, then the program has to contain an appropriately placed invocation of `#pragma STDC FENV_ACCESS ON` as shown.\(^{226}\)

### 7.6.2 The FENV_ROUND pragma

**Synopsis**

1

```c
#include <fenv.h>
#pragma STDC FENV_ROUND direction
#pragma STDC FENV_ROUND FE_DYNAMIC
```

**Description**

2 The `FENV_ROUND` pragma provides a means to specify a constant rounding direction for floating-point operations for standard floating types within a translation unit or compound statement. The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FENV_ROUND` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ROUND` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the static rounding mode is restored to its condition just before the compound statement. If this pragma is used in any other context, its behavior is undefined.

3 `direction` shall be one of the names of the supported rounding direction macros for operations for standard floating types (7.6), or `FE_DYNAMIC`. If any other value is specified, the behavior is undefined. If no `FENV_ROUND` pragma is in effect, or the specified constant rounding mode is `FE_DYNAMIC`, rounding is according to the mode specified by the dynamic floating-point environment, which is the dynamic rounding mode that was established either at thread creation or by a call to `fesetround`, `fesetmode`, `fesetenv`, or `feupdateenv`. If the `FE_DYNAMIC` mode is specified and `FENV_ACCESS` is “off”, the translator may assume that the default rounding mode is in effect.

4 The `FENV_ROUND` pragma affects operations for standard floating types. Within the scope of an `FENV_ROUND` pragma establishing a mode other than `FE_DYNAMIC`, floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic types to its semantic type, as done by classification macros), and invocations of functions indicated in the table below, for which macro replacement has not been suppressed (7.1.4), shall be evaluated according to the specified constant rounding mode (as though no constant mode was specified and the corresponding dynamic rounding mode had been established by a call to `fesetround`). Invocations of functions for which macro replacement has been suppressed and invocations of functions other than those indicated in the table below shall not be affected by constant rounding modes – they are affected by (and affect) only the dynamic mode. Floating constants (6.4.4.2) of a standard floating type that occur in the scope of a constant rounding mode shall be interpreted according to that mode.

- Functions affected by constant rounding modes – for standard floating types

\(^{226}\)The side effects impose a temporal ordering that requires two evaluations of $x + 1$. On the other hand, without the `#pragma STDC FENV_ACCESS ON` pragma, and assuming the default state is “off”, just one evaluation of $x + 1$ would suffice.
A function family listed in the table above indicates the functions for all standard floating types, where the function family is represented by the name of the functions without a suffix. For example, `acos` indicates the functions `acos`, `acosf`, and `acosl`.

5 NOTE Constant rounding modes (other than `FE_DYNAMIC`) could be implemented using dynamic rounding modes as illustrated in the following example:

```
{    
    #pragma STDC FENV_ROUND direction
    // compiler inserts:
    // #pragma STDC FENV_ACCESS ON
    // int __savedrdn;
    // __savedrdn = __swapround(direction);
    ... operations affected by constant rounding mode ...
    // compiler inserts:
    // __savedrdn = __swapround(__savedrdn);
    ... operations not affected by constant rounding mode ...
    // compiler inserts:
    // __savedrdn = __swapround(__savedrdn);
    ... operations affected by constant rounding mode ...
    // compiler inserts:
    // __swapround(__savedrdn);
}
```

where `__swapround` is defined by:

```
static inline int __swapround(const int new) {
    const int old = fegetround();
    fesetround(new);
    return old;
}
```

### 7.6.3 The FENV_DEC_ROUND pragma

#### Synopsis

```
#include <fenv.h>
#ifdef __STDC_IEC_60559_DFP__
#pragma STDC FENV_DEC_ROUND dec:direction
#endif
```
Description

The FENV_DEC_ROUND pragma is a decimal floating-point analog of the FENV_ROUND pragma. If 
FLT_RADIX is not 10, the FENV_DEC_ROUND pragma affects operators, functions, and floating 
constants only for decimal floating types. The affected functions are listed in the table below. 
If FLT_RADIX is 10, whether the FENV_ROUND and FENV_DEC_ROUND pragmas alter the rounding 
direction of both standard and decimal floating-point operations is implementation-defined. 
dec-direction shall be one of the decimal rounding direction macro names (FE_DEC_DOWNWARD, 
FE_DEC_TONEAREST, FE_DEC_TONEARESTFROMZERO, FE_DEC_TOWARDZERO, and 
FE_DEC_UPWARD) defined in 7.6, to specify a constant rounding mode, or FE_DEC_DYNAMIC, to 
specify dynamic rounding. The corresponding dynamic rounding mode can be established by a call to 
fe_dec_setround.

Functions affected by constant rounding modes – for decimal floating types

<table>
<thead>
<tr>
<th>Header</th>
<th>Function families</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;math.h&gt;</td>
<td>acos, acospi, asin, asinpi, atan, atan2, atan2pi, atanpi</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>cos, cospi, sin, sinpi, tan, tanpi</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>acosh, asinh, atanh</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>cosh, sinh, tanh</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>exp, exp10, exp10m, exp2, exp2m, expm</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>log, log10, log10m, log1p, log1p, log2, log2p, log1p</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>scalbn, scalbln, ldexp</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>cbrt, compoundn, hypot, pow, powm, powr, rootn, rsqrt, sqrt</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>erf, erfc</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>lgamma, tgamma</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>rint, nearbyint, lrint, lllrint</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>quantize</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>fdim</td>
</tr>
<tr>
<td>&lt;math.h&gt;</td>
<td>fma</td>
</tr>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>d32add, d64add, d32sub, d64sub, d32mul, d64mul, d32div, d64div,</td>
</tr>
<tr>
<td></td>
<td>d32fma, d64fma, d32sqrt, d64sqrt</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>strfromstrto</td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>printf and scanf families</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>wprintf and wscanf families</td>
</tr>
</tbody>
</table>

A function family listed in the table above indicates the functions for all decimal floating types, 
where the function family is represented by the name of the functions without a suffix. For example, 
acos indicates the functions acos32, acos64, andacosd128.

7.6.4 Floating-point exceptions

The following functions provide access to the floating-point status flags.\(^{227}\) The int input argument 
for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise 
OR of one or more floating-point exception macros, for example FE_OVERFLOW | FE_INEXACT. For 
other argument values, the behavior of these functions is undefined.

7.6.4.1 The feclearexcept function

Synopsis

\(^{227}\) The functions fetestexcept, feraiseexcept, and feclearexcept support the basic abstraction of flags that are either set or clear. An implementation can endow floating-point status flags with more information — for example, the address of 
the code which first raised the floating-point exception; the functions fegetexceptflag and fesetexceptflag deal with 
the full content of flags.
#include <fenv.h>
int feclearexcept(int excepts);

Description
The **feclearexcept** function attempts to clear the supported floating-point exceptions represented by its argument.

Returns
The **feclearexcept** function returns zero if the **excepts** argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.

### 7.6.4.2 The **fegetexceptflag** function

**Synopsis**

```c
#include <fenv.h>
int fegetexceptflag(fexcept_t *flagp, int excepts);
```

Description
The **fegetexceptflag** function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument **excepts** in the object pointed to by the argument **flagp**.

Returns
The **fegetexceptflag** function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.

### 7.6.4.3 The **feraiseexcept** function

**Synopsis**

```c
#include <fenv.h>
int feraiseexcept(int excepts);
```

Description
The **feraiseexcept** function attempts to raise the supported floating-point exceptions represented by its argument. The order in which these floating-point exceptions are raised is unspecified, except as stated in F.8.6. Whether the **feraiseexcept** function additionally raises the “inexact” floating-point exception whenever it raises the “overflow” or “underflow” floating-point exception is implementation-defined.

Returns
The **feraiseexcept** function returns zero if the **excepts** argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.

### 7.6.4.4 The **fesetexcept** function

**Synopsis**

```c
#include <fenv.h>
int fesetexcept(int excepts);
```

Description
The **fesetexcept** function attempts to set the supported floating-point exception flags represented by its argument. This function does not clear any floating-point exception flags. This function changes the state of the floating-point exception flags, but does not cause any other side effects that might be associated with raising floating-point exceptions. The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken. The specification in F:8.6 is in the same spirit. Enabled traps for floating-point exceptions are not taken.
The `fesetexceptflag` function attempts to set the floating-point status flags indicated by the argument `excepts` to the states stored in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. This function does not raise floating-point exceptions, but only sets the state of the flags.

**Returns**

3. The `fesetexceptflag` function returns zero if the `excepts` argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.

### 7.6.4.6 The `fetestexceptflag` function

**Synopsis**

```
#include <fenv.h>
int fetestexceptflag(const fexcept_t *flagp, int excepts);
```

**Description**

2. The `fetestexceptflag` function determines which of a specified subset of the floating-point exception flags are set in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag` whose second argument represented at least those floating-point exceptions represented by the argument `excepts`. The `excepts` argument specifies the floating-point status flags to be queried.

**Returns**

3. The `fetestexceptflag` function returns the value of the bitwise OR of the floating-point exception macros included in `excepts` corresponding to the floating-point exceptions set in `*flagp`.

### 7.6.4.7 The `fetestexcept` function

**Synopsis**

```
#include <fenv.h>
int fetestexcept(int excepts);
```

**Description**

2. The `fetestexcept` function determines which of a specified subset of the floating-point exception flags are currently set. The `excepts` argument specifies the floating-point status flags to be queried.\(^{230}\)

**Returns**

3. The `fetestexcept` function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.

**EXAMPLE**

Call `f` if “invalid” is set, then `g` if “overflow” is set:

```c
#include <fenv.h>
/* ... */
```

\(^{230}\)This mechanism allows testing several floating-point exceptions with just one function call.
{  
    #pragma STDC FENV_ACCESS ON
    int set_excepts;
    feclearexcept(FE_INVALID | FE_OVERFLOW);
    // maybe raise exceptions
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
    if (set_excepts & FE_INVALID) f();
    if (set_excepts & FE_OVERFLOW) g();
    /* ... */
}

7.6.5 Rounding and other control modes

The fegetround and fesetround functions provide control of rounding direction modes. The fegetmode and fesetmode functions manage all the implementation’s dynamic floating-point control modes collectively.

7.6.5.1 The fegetmode function

Synopsis

```c
#include <fenv.h>
int fegetmode(femode_t *modep);
```

Description

The fegetmode function attempts to store all the dynamic floating-point control modes in the object pointed to by modep.

Returns

The fegetmode function returns zero if the modes were successfully stored. Otherwise, it returns a nonzero value.

7.6.5.2 The fegetround function

Synopsis

```c
#include <fenv.h>
int fegetround(void);
```

Description

The fegetround function gets the current rounding direction value of the dynamic rounding direction mode.

Returns

The fegetround function returns the value of the rounding direction macro representing the current dynamic rounding direction or a negative value if there is no such rounding direction macro or the current dynamic rounding direction is not determinable.

7.6.5.3 The fe_dec_getround function

Synopsis

```c
#include <fenv.h>
#ifdef __STDC_IEC_60559_DFP__
int fe_dec_getround(void);
#endif
```

Description

The fe_dec_getround function gets the current value of the dynamic rounding direction mode for decimal floating-point operations.

Returns

§ 7.6.5.3 Library 179
The `fe_dec_getround` function returns the value of the rounding direction macro representing the current dynamic rounding direction for decimal floating-point operations, or a negative value if there is no such rounding macro or the current rounding direction is not determinable.

### 7.6.5.4 The `fesetmode` function

**Synopsis**

```c
#include <fenv.h>
int fesetmode(const femode_t *modep);
```

**Description**

The `fesetmode` function attempts to establish the dynamic floating-point modes represented by the object pointed to by `modep`. The argument `modep` shall point to an object set by a call to `fegetmode`, or equal `FE_DFL_MODE` or a dynamic floating-point mode state macro defined by the implementation.

**Returns**

The `fesetmode` function returns zero if the modes were successfully established. Otherwise, it returns a nonzero value.

### 7.6.5.5 The `fesetround` function

**Synopsis**

```c
#include <fenv.h>
int fesetround(int round);
```

**Description**

The `fesetround` function establishes the rounding direction represented by its argument `round`. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

**Returns**

The `fesetround` function returns zero if and only if the requested rounding direction was established. Dynamic rounding direction mode was set to the requested rounding direction.

**EXAMPLE**

Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```c
#include <fenv.h>
#include <assert.h>

void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

### 7.6.5.6 The `fe_dec_setround` function

**Synopsis**

```c
#include <fenv.h>
ifdef __STDC_IEC_60559_DFP__
int fe_dec_setround(int round);
endif
```

180 Library § 7.6.5.6
Description
2 The `fe_dec_setround` function sets the dynamic rounding direction mode for decimal floating-point operations to be the rounding direction represented by its argument `round`. If the argument is not equal to the value of a decimal rounding direction macro, the rounding direction is not changed.

3 If `FLT_RADIX` is not 10, the rounding direction altered by the `fesetround` function is independent of the rounding direction altered by the `fe_dec_setround` function; otherwise if `FLT_RADIX` is 10, whether the `fesetround` and `fe_dec_setround` functions alter the rounding direction of both standard and decimal floating-point operations is implementation-defined.

Returns
4 The `fe_dec_setround` function returns a zero value if and only if the argument is equal to a decimal rounding direction macro (that is, if and only if the dynamic rounding direction mode for decimal floating-point operations was set to the requested rounding direction).

7.6.6 Environment
1 The functions in this section manage the floating-point environment — status flags and control modes — as one entity.

7.6.6.1 The `fegetenv` function
Synopsis
1
```c
#include <fenv.h>
int fegetenv(fenv_t *envp);
```

Description
2 The `fegetenv` function attempts to store the current dynamic floating-point environment in the object pointed to by `envp`.

Returns
3 The `fegetenv` function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.

7.6.6.2 The `feholdexcept` function
Synopsis
1
```c
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description
2 The `feholdexcept` function saves the current dynamic floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and then installs a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.\(^{231}\)

Returns
3 The `feholdexcept` function returns zero if and only if non-stop floating-point exception handling was successfully installed.

7.6.6.3 The `fesetenv` function
Synopsis
1
```c
#include <fenv.h>
int fesetenv(const fenv_t *envp);
```

\(^{231}\)IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the `feholdexcept` function can be used in conjunction with the `feupdateenv` function to write routines that hide spurious floating-point exceptions from their callers.

§ 7.6.6.3 Library 181
Description

The `fesetenv` function attempts to establish the dynamic floating-point environment represented by the object pointed to by `envp`. The argument `envp` shall point to an object set by a call to `fegetenv` or `feholdexcept`, or equal a dynamic floating-point environment macro. Note that `fesetenv` merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.

Returns

The `fesetenv` function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.

7.6.6.4 The `feupdateenv` function

Synopsis

```c
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

Description

The `feupdateenv` function attempts to save the currently raised floating-point exceptions in its automatic storage, install the dynamic floating-point environment represented by the object pointed to by `envp`, and then raise the saved floating-point exceptions. The argument `envp` shall point to an object set by a call to `feholdexcept` or `fegetenv`, or equal a dynamic floating-point environment macro.

Returns

The `feupdateenv` function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.

EXAMPLE

Hide spurious underflow floating-point exceptions:

```c
#include <fenv.h>
double f(double x)
{
    #pragma STDC FENV_ACCESS ON
    double result;
    fenv_t save_env;
    if (feholdexcept(&save_env))
        return /* indication of an environmental problem */;
    // compute result
    if (/* test spurious underflow */) // change to any other test that you wish
        if (feclearexcept(FE_UNDERFLOW))
            return /* indication of an environmental problem */;
    if (feupdateenv(&save_env))
        return /* indication of an environmental problem */;
    return result;
}
```
7.7 Characteristics of floating types <float.h>

1 The header <float.h> defines several macros that expand to various limits and parameters of the standard floating-point real floating types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in §5.2.4.2.2, §5.2.4.2.2 and §5.2.4.2.3. A summary is given in Annex E.
7.8 Format conversion of integer types `<inttypes.h>`

The header `<inttypes.h>` includes the header `<stdint.h>` and extends it with additional facilities provided by hosted implementations.

It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the type

```
imaxdiv_t
```

which is a structure type that is the type of the value returned by the `imaxdiv` function. For each type declared in `<stdint.h>`, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.\(^{232}\)

**Forward references:** integer types `<stdint.h>` (7.20), formatted input/output functions (7.21.6), formatted wide character input/output functions (7.29.2).

7.8.1 Macros for format specifiers

Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of `PRI` (character string literals for the `fprintf` and `fwprintf` family) or `SCN` (character string literals for the `fscanf` and `fwscanf` family),\(^{233}\) followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.20.1. In these names, `N` represents the width of the type as described in 7.20.1. For example, `PRIdFAST32` can be used in a format string to print the value of an integer of type `int_fast32_t`.

The `fprintf` macros for signed integers are:

```
PRIdN  PRIdLEASTN  PRIdFASTN  PRIdMAX  PRIdPTR
PRiN   PRIiLEASTN  PRIiFASTN  PRIiMAX  PRIiPTR
```

The `fprintf` macros for unsigned integers are:

```
PRIoN  PRIoLEASTN  PRIoFASTN  PRIoMAX  PRIoPTR
PRIuN  PRIuLEASTN  PRIuFASTN  PRIuMAX  PRIuPTR
PRIxN  PRIxLEASTN  PRIxFASTN  PRIxMAX  PRIxPTR
PRIXN  PRIXLEASTN  PRIXFASTN  PRIXMAX  PRIXPTR
```

The `fscanf` macros for signed integers are:

```
SCNdN  SCNdLEASTN  SCNdFASTN  SCNdMAX  SCNdPTR
SCNiN  SCNiLEASTN  SCNiFASTN  SCNiMAX  SCNiPTR
```

The `fscanf` macros for unsigned integers are:

```
SCNoN  SCNoLEASTN  SCNoFASTN  SCNoMAX  SCNoPTR
SCNuN  SCNuLEASTN  SCNuFASTN  SCNuMAX  SCNuPTR
SCNxN  SCNxLEASTN  SCNxFASTN  SCNxMAX  SCNxPTR
```

For each type that the implementation provides in `<stdint.h>`, the corresponding `fprintf` macros shall be defined and the corresponding `fscanf` macros shall be defined unless the implementation does not have a suitable `fscanf` length modifier for the type.

**EXAMPLE**

```c
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
    uintmax_t i = UINTMAX_MAX; // this type always exists
    wprintf(L"The largest integer value is %020"
```

\(^{232}\)See “future library directions” (7.31.6).

\(^{233}\)Separate macros are given for use with `fprintf` and `fscanf` functions because, in the general case, different format specifiers might be required for `fprintf` and `fscanf`, even when the type is the same.
7.8.2 Functions for greatest-width integer types

7.8.2.1 The imaxabs function

Synopsis

```c
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

Description

The `imaxabs` function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.\(^{234}\)

Returns

The `imaxabs` function returns the absolute value.

7.8.2.2 The imaxdiv function

Synopsis

```c
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

The `imaxdiv` function computes \(\text{num}er / \text{denom}\) and \(\text{num}er \% \text{denom}\) in a single operation.

Returns

The `imaxdiv` function returns a structure of type `imaxdiv_t` comprising both the quotient and the remainder. The structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type `intmax_t`. If either part of the result cannot be represented, the behavior is undefined.

7.8.2.3 The strtoimax and strtoumax functions

Synopsis

```c
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
```

Description

The `strtoimax` and `strtoumax` functions are equivalent to the `strtol`, `strtoll`, `strtoul`, and `strtoull` functions, except that the initial portion of the string is converted to `intmax_t` and `uintmax_t` representation, respectively.

Returns

The `strtoimax` and `strtoumax` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `INTMAX_MAX`, `INTMAX_MIN`, or `UINTMAX_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

Forward references: the `strtol`, `strtoll`, `strtoul`, and `strtoull` functions (7.22.1.7).

7.8.2.4 The wcstolmax and wcstoumax functions

\(^{234}\)The absolute value of the most negative number cannot be represented in two’s complement.
Synopsis

1  
#include <stddef.h>  // for wchar_t
#include <inttypes.h>

intmax_t wcstoismax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);

Description

2  The wcstoismax and wcstoumax functions are equivalent to the wcstol, wcstoll, wcstoul, and wcstoull functions except that the initial portion of the wide string is converted to intmax_t and uintmax_t representation, respectively.

Returns

3  The wcstoismax function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, INTMAX_MAX, INTMAX_MIN, or UINTMAX_MAX is returned (according to the return type and sign of the value, if any), and the value of the macro ERANGE is stored in errno.

Forward references: the wcstol, wcstoll, wcstoul, and wcstoull functions (7.29.4.1.3).
7.9 Alternative spellings <iso646.h>

The header <iso646.h> defines the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

```c
and &&
and_eq &=
bitand &
bitor |
compl ~
not !
not_eq !=
or ||
or_eq ==
xor ^
xor_eq ^=
```
7.10 Sizes of integer types <limits.h>

1 The header <limits.h> defines several macros that expand to various limits and parameters of the standard integer types.

2 The macros, their meanings, and the constraints (or restrictions) on their values are listed in 5.2.4.2.1. A summary is given in Annex E.
7.11 Localization `<locale.h>`

The header `<locale.h>` declares two functions, one type, and defines several macros.

The type is

```c
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained in 7.11.2.1. In the “C” locale, the members shall have the values specified in the comments.

```c
char *decimal_point; // "."  
char *thousands_sep; // ""  
char *grouping; // ""  
char *mon_decimal_point; // ""  
char *mon_thousands_sep; // ""  
char *mon_grouping; // ""  
char *positive_sign; // ""  
char *negative_sign; // ""  
char *currency_symbol; // ""  
char frac_digits; // CHAR_MAX  
char p_cs_precedes; // CHAR_MAX  
char n_cs_precedes; // CHAR_MAX  
char p_sep_by_space; // CHAR_MAX  
char n_sep_by_space; // CHAR_MAX  
char p_sign_posn; // CHAR_MAX  
char n_sign_posn; // CHAR_MAX  
char *int_curr_symbol; // ""  
char int_frac_digits; // CHAR_MAX  
char int_p_cs_precedes; // CHAR_MAX  
char int_n_cs_precedes; // CHAR_MAX  
char int_p_sep_by_space; // CHAR_MAX  
char int_n_sep_by_space; // CHAR_MAX  
char int_p_sign_posn; // CHAR_MAX  
char int_n_sign_posn; // CHAR_MAX
```

The macros defined are `NULL` (described in 7.19); and

```c
LC_ALL  
LC_COLLATE  
LC_CTYPE  
LC_MONETARY  
LC_NUMERIC  
LC_TIME
```

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the `setlocale` function. Additional macro definitions, beginning with the characters `LC_` and an uppercase letter, may also be specified by the implementation.

7.11.1 Locale control

7.11.1.1 The `setlocale` function

**Synopsis**

```c
#include <locale.h>
char *setlocale(int category, const char *locale);
```
Description

The `setlocale` function selects the appropriate portion of the program’s locale as specified by the `category` and `locale` arguments. The `setlocale` function may be used to change or query the program’s entire current locale or portions thereof. The value `LC_ALL` for `category` names the program’s entire locale; the other values for `category` name only a portion of the program’s locale. `LC_COLLATE` affects the behavior of the `strcoll` and `strxfrm` functions. `LC_CTYPE` affects the behavior of the character handling functions\(^{237}\) and the multibyte and wide character functions. `LC_MONETARY` affects the monetary formatting information returned by the `localeconv` function. `LC_NUMERIC` affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the `localeconv` function. `LC_TIME` affects the behavior of the `strftime` and `wcsftime` functions.

A value of "C" for `locale` specifies the minimal environment for C translation; a value of "" for `locale` specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to `setlocale`.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

A call to the `setlocale` function may introduce a data race with other calls to the `setlocale` function or with calls to functions that are affected by the current locale. The implementation shall behave as if no library function calls the `setlocale` function.

Returns

If a pointer to a string is given for `locale` and the selection can be honored, the `setlocale` function returns a pointer to the string associated with the specified `category` for the new locale. If the selection cannot be honored, the `setlocale` function returns a null pointer and the program’s locale is not changed.

A null pointer for `locale` causes the `setlocale` function to return a pointer to the string associated with the `category` for the program’s current locale; the program’s locale is not changed.\(^{238}\)

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category will restore that part of the program’s locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `setlocale` function.

**Forward references:** formatted input/output functions (7.21.6), multibyte/wide character conversion functions (7.22.7), multibyte/wide string conversion functions (7.22.8), numeric conversion functions (7.22.1), the `strcoll` function (7.24.4.3), the `strftime` function (7.27.3.5), the `strxfrm` function (7.24.4.5).

### 7.11.2 Numeric formatting convention inquiry

#### 7.11.2.1 The `localeconv` function

**Synopsis**

```
#include <locale.h>
struct lconv *localeconv(void);
```

**Description**

The `localeconv` function sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

\(^{237}\)The only functions in 7.4 whose behavior is not affected by the current locale are `isdigit` and `isxdigit`.

\(^{238}\)The implementation is thus required to arrange to encode in a string the various categories due to a heterogeneous locale when `category` has the value `LC_ALL`.
The members of the structure with type `char *` are pointers to strings, any of which (except `decimal_point`) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. Apart from `grouping` and `mon_grouping`, the strings shall start and end in the initial shift state. The members with type `char` are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The members include the following:

- **char *decimal_point**
  The decimal-point character used to format nonmonetary quantities.

- **char *thousands_sep**
  The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

- **char *grouping**
  A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

- **char *mon_decimal_point**
  The decimal-point used to format monetary quantities.

- **char *mon_thousands_sep**
  The separator for groups of digits before the decimal-point in formatted monetary quantities.

- **char *mon_grouping**
  A string whose elements indicate the size of each group of digits in formatted monetary quantities.

- **char *positive_sign**
  The string used to indicate a nonnegative-valued formatted monetary quantity.

- **char *negative_sign**
  The string used to indicate a negative-valued formatted monetary quantity.

- **char *currency_symbol**
  The local currency symbol applicable to the current locale.

- **char frac_digits**
  The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

- **char p_cs_precedes**
  Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

- **char n_cs_precedes**
  Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

- **char p_sep_by_space**
  Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a nonnegative locally formatted monetary quantity.

- **char n_sep_by_space**
  Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative locally formatted monetary quantity.

- **char p_sign_posn**
  Set to a value indicating the positioning of the `positive_sign` for a nonnegative locally formatted monetary quantity.
char n_sign_posn
Set to a value indicating the positioning of the negative_sign for a negative locally formatted monetary quantity.

char *int_curr_symbol
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char int_frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char int_p_cs_precedes
Set to 1 or 0 if the int_curr_symbol respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

char int_n_cs_precedes
Set to 1 or 0 if the int_curr_symbol respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

char int_p_sep_by_space
Set to a value indicating the separation of the int_curr_symbol, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

char int_n_sep_by_space
Set to a value indicating the separation of the int_curr_symbol, the sign string, and the value for a negative internationally formatted monetary quantity.

char int_p_sign_posn
Set to a value indicating the positioning of the positive_sign for a nonnegative internationally formatted monetary quantity.

char int_n_sign_posn
Set to a value indicating the positioning of the negative_sign for a negative internationally formatted monetary quantity.

4 The elements of grouping and mon_grouping are interpreted according to the following:

CHAR_MAX  No further grouping is to be performed.
0  The previous element is to be repeatedly used for the remainder of the digits.
other  The integer value is the number of digits that compose the current group. The next element is examined to determine the size of the next group of digits before the current group.

5 The values of p_sep_by_space, n_sep_by_space, int_p_sep_by_space, and int_n_sep_by_space are interpreted according to the following:

0  No space separates the currency symbol and value.
1  If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.
2  If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.
For \textit{int\_p\_sep\_by\_space} and \textit{int\_n\_sep\_by\_space}, the fourth character of \textit{int\_curr\_symbol} is used instead of a space.

6 The values of \textit{p\_sign\_posn}, \textit{n\_sign\_posn}, \textit{int\_p\_sign\_posn}, and \textit{int\_n\_sign\_posn} are interpreted according to the following:

0 Parentheses surround the quantity and currency symbol.

1 The sign string precedes the quantity and currency symbol.

2 The sign string succeeds the quantity and currency symbol.

3 The sign string immediately precedes the currency symbol.

4 The sign string immediately succeeds the currency symbol.

7 The implementation shall behave as if no library function calls the \texttt{localeconv} function.

8 The \texttt{localeconv} function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the \texttt{localeconv} function. In addition, calls to the \texttt{setlocale} function with categories \texttt{LC\_ALL}, \texttt{LC\_MONETARY}, or \texttt{LC\_NUMERIC} may overwrite the contents of the structure.

EXAMPLE 1 The following table illustrates rules which might well be used by four countries to format monetary quantities.

<table>
<thead>
<tr>
<th>Country</th>
<th>Positive</th>
<th>Negative</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country1</td>
<td>1.234,56 mk</td>
<td>-1.234,56 mk</td>
<td>FIM 1.234,56</td>
<td>FIM -1.234,56</td>
</tr>
<tr>
<td>Country2</td>
<td>L.1.234</td>
<td>-L.1.234</td>
<td>ITL 1.234</td>
<td>ITL 1.234</td>
</tr>
<tr>
<td>Country3</td>
<td>ƒ1.234,56</td>
<td>ƒ-1.234,56</td>
<td>NLG 1.234,56</td>
<td>NLG -1.234,56</td>
</tr>
<tr>
<td>Country4</td>
<td>SFr.1.234,56</td>
<td>SFr.1.234,56C</td>
<td>CHF 1,234.56</td>
<td>CHF 1,234.56C</td>
</tr>
</tbody>
</table>

10 For these four countries, the respective values for the monetary members of the structure returned by \texttt{localeconv} could be:

<table>
<thead>
<tr>
<th>Country1</th>
<th>Country2</th>
<th>Country3</th>
<th>Country4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mon_decimal_point</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
</tr>
<tr>
<td>mon_thousands_sep</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>mon_grouping</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td>positive_sign</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>negative_sign</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
</tr>
<tr>
<td>currency_symbol</td>
<td>&quot;mk&quot;</td>
<td>&quot;L.&quot;</td>
<td>&quot;\u0192&quot;</td>
</tr>
<tr>
<td>frac_digits</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>p_cs_precedes</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_cs_precedes</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p_sep_by_space</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n_sep_by_space</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>p_sign_posn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_sign_posn</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>int_curr_symbol</td>
<td>&quot;FIM&quot;</td>
<td>&quot;ITL&quot;</td>
<td>&quot;NLG&quot;</td>
</tr>
<tr>
<td>int_frac_digits</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>int_p_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int_n_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int_p_sep_by_space</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int_n_sep_by_space</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>int_p_sign_posn</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int_n_sign_posn</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

§ 7.11.2.1 Library
EXAMPLE 2 The following table illustrates how the `cs_precedes`, `sep_by_space`, and `sign_posn` members affect the formatted value.

<table>
<thead>
<tr>
<th><code>p_cs_precedes</code></th>
<th><code>p_sign_posn</code></th>
<th><code>p_sep_by_space</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(1.25$)</td>
</tr>
<tr>
<td>1</td>
<td>+1.25$</td>
<td>+ 1.25$</td>
</tr>
<tr>
<td>2</td>
<td>1.25$+</td>
<td>1.25 $+</td>
</tr>
<tr>
<td>3</td>
<td>1.25+$</td>
<td>1.25 +$</td>
</tr>
<tr>
<td>4</td>
<td>1.25$+</td>
<td>1.25 $+</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>+$1.25</td>
</tr>
<tr>
<td>1</td>
<td>+$1.25</td>
<td>+$ 1.25</td>
</tr>
<tr>
<td>2</td>
<td>=$1.25</td>
<td>=$ 1.25</td>
</tr>
<tr>
<td>3</td>
<td>+$1.25</td>
<td>+$ 1.25</td>
</tr>
<tr>
<td>4</td>
<td>+=1.25</td>
<td>$+ 1.25</td>
</tr>
</tbody>
</table>

194 Library § 7.11.2.1
7.12 Mathematics <math.h>

The header <math.h> declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more `double` parameters, a `double` return value, or both; and other functions with the same name but with `f` and `l` suffixes, which are corresponding functions with `float` and `long double` parameters, return values, or both.\(^{239}\) Integer arithmetic functions and conversion functions are discussed later.

The feature test macro `__STDC_VERSION_MATH_H__` expands to the token `yyyymmL`.

The types `float_t` and `double_t` are floating types at least as wide as `float` and `double`, respectively, and such that `double_t` is at least as wide as `float_t`. If `FLT_EVAL_METHOD` equals 0, `float_t` and `double_t` are `float` and `double`, respectively; if `FLT_EVAL_METHOD` equals 1, they are both `double`; if `FLT_EVAL_METHOD` equals 2, they are both `long double`; and for other values of `FLT_EVAL_METHOD`, they are otherwise implementation-defined.\(^{240}\)

The types `__Decimal32_t` and `__Decimal64_t` are decimal floating types at least as wide as `__Decimal32` and `__Decimal64`, respectively, and such that `__Decimal64_t` is at least as wide as `__Decimal32_t`. If `DEC_EVAL_METHOD` equals 0, `__Decimal32_t` and `__Decimal64_t` are `__Decimal32` and `__Decimal64`, respectively; if `DEC_EVAL_METHOD` equals 1, they are both `__Decimal64`; if `DEC_EVAL_METHOD` equals 2, they are both `__Decimal128`, and for other values of `DEC_EVAL_METHOD`, they are otherwise implementation-defined.

The macro `HUGE_VAL` expands to a positive `double` constant expression, not necessarily representable as a `float`. The macros `HUGE_VALF` and `HUGE_VALL` are respectively `float` and `long double` analogs of `HUGE_VAL`.\(^{241}\)

The macro `HUGE_VAL_D32` expands to a constant expression of type `__Decimal32_t` representing positive infinity. The macros `HUGE_VAL_D64` and `HUGE_VAL_D128` can be positive infinities in an implementation that supports infinities.

\(^{239}\)Particularly on systems with wide expression evaluation, a `<math.h>` function might pass arguments and return values in wider format than the synopsis prototype indicates.

\(^{240}\)The types `float_t` and `double_t` are intended to be the implementation’s most efficient types at least as wide as `float` and `double`, respectively. For `FLT_EVAL_METHOD` equal 0, 1, or 2, the type `float_t` is the narrowest type used by the implementation to evaluate floating expressions.

\(^{241}\)`HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` can be positive infinities in an implementation that supports infinities.
are respectively _Decimal64 and _Decimal128 analogs of HUGE_VAL_D32._

7 The macro

```
INFINITY
```

expands to a constant expression of type `float` representing positive or unsigned infinity, if available; else to a positive constant of type `float` that overflows at translation time.\(^{242}\)

8 The macro

```
DEC_INFINITY
```

expands to a constant expression of type _Decimal32 representing positive infinity._

9 The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the `float` type. It expands to a constant expression of type `float` representing a quiet NaN.

10 The macro

```
DEC_NAN
```

expands to a constant expression of type _Decimal32 representing a quiet NaN._

11 The signaling NaN macros

```
SNANF
SNAN
SNANL
```

each is defined if and only if the respective type contains signaling NaNs (5.2.4.2.2). They expand to a constant expression of the respective type representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

12 The decimal signaling NaN macros

```
SNAND32
SNAND64
SNAND128
```

each expands to a constant expression of the respective decimal floating type representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

13 The number classification macros

```
FP_INFINITY
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with `FP_` and an uppercase letter, may also be specified by the implementation.

\(^{242}\)In this case, using `INFINITY` will violate the constraint in 6.4.4 and thus require a diagnostic.
The math rounding direction macros

The macros

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

represent the rounding directions of the functions `ceil`, `floor`, `trunc`, `round`, and `roundeven`, respectively, that convert to integral values in floating-point formats. They expand to integer constant expressions with distinct values suitable for use as the second argument to the `fromfp`, `ufromfp`, `fromfpx`, and `ufromfpx` functions.

The macro

```
FP_FAST_FMA
```

is optionally defined. If defined, it indicates that the `fma` function generally executes about as fast as, or faster than, a multiply and an add of `double` operands.\(^{243}\) The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, `float` and `long double` analogs of `FP_FAST_FMA`. If defined, these macros expand to the integer constant 1.

The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

are, respectively, `_Decimal32`, `_Decimal64`, and `_Decimal128` analogs of `FP_FAST_FMA`.

Each of the macros

```
FP_FAST_FADD  FP_FAST_DSUBL  FP_FAST_FDIVL  FP_FAST_FFMA
FP_FAST_FADDL  FP_FAST_FMUL  FP_FAST_DDIVL  FP_FAST_FFMAL
FP_FAST_FADDDL  FP_FAST_FMULL  FP_FAST_FSQRT  FP_FAST_DFMA
FP_FAST_FSUBL  FP_FAST_DMULL  FP_FAST_FSQRTL
FP_FAST_FSUBL  FP_FAST_FDIV  FP_FAST_DSQRTL
```

is optionally defined. If defined, it indicates that the corresponding function generally executes about as fast, or faster, than the corresponding operation or function of the argument type with result type the same as the argument type followed by conversion to the narrower type. For `FP_FAST_FMA`, `FP_FAST_FFMAL`, and `FP_FAST_DFMA`, the comparison is to a call to `fma` or `fmal` followed by a conversion, not to separate multiply, add, and conversion. If defined, these macros expand to the integer constant 1.

The macros

```
FP_FAST_D32ADD64  FP_FAST_D32MULD64  FP_FAST_D32FMAD64
FP_FAST_D32ADD128  FP_FAST_D32MULD128  FP_FAST_D32FMAD128
FP_FAST_D64ADD128  FP_FAST_D64MULD128  FP_FAST_D64FMAD128
FP_FAST_D32SUBD64  FP_FAST_D32DIVD64  FP_FAST_D32SQRTD64
FP_FAST_D32SUBD128  FP_FAST_D32DIVD128  FP_FAST_D32SQRTD128
FP_FAST_D64SUBD128  FP_FAST_D64DIVD128  FP_FAST_D64SQRTD128
```

\(^{243}\) Typically, the `FP_FAST_FMA` macro is defined if and only if the `fma` function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.
are analogs of FP_FAST_FADDL, FP_FAST_FADDL, FP_FAST_DADDL, etc., for decimal floating types.

The macros

```c
FP_ILOGB0
FP_ILOGBNAN
```

expand to integer constant expressions whose values are returned by `ilogb(x)` if `x` is zero or NaN, respectively. The value of `FP_ILOGB0` shall be either `INT_MIN` or `-INT_MAX`. The value of `FP_ILOGBNAN` shall be either `INT_MAX` or `INT_MIN`.

The macros

```c
FP_LLOGB0
FP_LLOGBNAN
```

expand to integer constant expressions whose values are returned by `llogb(x)` if `x` is zero or NaN, respectively. The value of `FP_LLOGB0` shall be `LONG_MIN` if the value of `FP_ILOGB0` is `INT_MIN`, and shall be `-LONG_MAX` if the value of `FP_ILOGB0` is `-INT_MAX`. The value of `FP_LLOGBNAN` shall be `LONG_MAX` if the value of `FP_ILOGBNAN` is `INT_MAX`, and shall be `LONG_MIN` if the value of `FP_ILOGBNAN` is `INT_MIN`.

The macros

```c
MATH_ERRNO
MATH_ERREXCEPT
```

expand to the integer constants 1 and 2, respectively; the macro

```c
math_errhandling
```

expands to an expression that has type `int` and the value `MATH_ERRNO`, `MATH_ERREXCEPT`, or the bitwise OR of both. The value of `math_errhandling` is constant for the duration of the program. It is unspecified whether `math_errhandling` is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name `math_errhandling`, the behavior is undefined. If the expression `math_errhandling & MATH_ERREXCEPT` can be nonzero, the implementation shall define the macros `FE_DIVBYZERO`, `FE_INVALID`, and `FE_OVERFLOW` in `<fenv.h>`.

### 7.12.1 Treatment of error conditions

The behavior of each of the functions in `<math.h>` is specified for all representable values of its input arguments, except where explicitly stated otherwise. Each function shall execute as if it were a single operation without raising `SIGFPE` and without generating any of the floating-point exceptions “invalid”, “divide-by-zero”, or “overflow” except to reflect the result of the function.

For all functions, a domain error occurs if and only if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.241) Whether a signaling NaN input causes a domain error is implementation-defined. On a domain error, the function returns an implementation-defined value; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the integer expression `errno` acquires the value `EDOM`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the “invalid” floating-point exception is raised.

Similarly, a pole error (also known as a singularity or infinitary) occurs if and only if the mathematical function has an exact infinite result as the finite input argument(s) are approached in the limit (for example, `log(0.0)`). The description of each function lists any required pole errors; an implementation may define additional pole errors, provided that such errors are consistent with the mathematical

241) In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.
definition of the function. On a pole error, the function returns an implementation-defined value; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the “divide-by-zero” floating-point exception is raised.

Likewise, a range error occurs if and only if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude. The description of each function lists any required range errors; an implementation may define additional range errors, provided that such errors are consistent with the mathematical definition of the function and are the result of either overflow or underflow.

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the function returns the value of the macro `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL` according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression `errno` acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the “overflow” floating-point exception is raised.

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether `errno` acquires the value `ERANGE` is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the “underflow” floating-point exception is raised is implementation-defined.

If a domain, pole, or range error occurs and the integer expression `math_errhandling & MATH_ERRNO` is zero, then `errno` shall either be set to the value corresponding to the error or left unmodified. If no such error occurs, `errno` shall be left unmodified regardless of the setting of `math_errhandling`.

### 7.12.2 The `FP_CONTRACT` pragma

#### Synopsis

```c
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

#### Description

The `FP_CONTRACT` pragma can be used to allow (if the state is “on”) or disallow (if the state is “off”) the implementation to contract expressions (6.5). Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state (“on” or “off”) for the pragma is implementation-defined.

#### 7.12.3 Classification macros

In the synopses in this subclause, `real-floating` indicates that the argument shall be an expression of real floating type.

### 7.12.3.1 The `fpclassify` macro

245) The term underflow here is intended to encompass both “gradual underflow” as in IEC 60559 and also “flush-to-zero” underflow.

246) Math errors are being indicated by the floating-point exception flags rather than by `errno`.
Synopsis
1

#include <math.h>
int fpclassify(real-floating x);

Description
2
The fpclassify macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.247)

Returns
3
The fpclassify macro returns the value of the number classification macro appropriate to the value of its argument.

7.12.3.2 The iscanonical macro
Synopsis
1

#include <math.h>
int iscanonical(real-floating x);

Description
2
The iscanonical macro determines whether its argument value is canonical (5.2.4.2.2). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then, determination is based on the type of the argument.

Returns
3
The iscanonical macro returns a nonzero value if and only if its argument is canonical.

7.12.3.3 The isfinite macro
Synopsis
1

#include <math.h>
int isfinite(real-floating x);

Description
2
The isfinite macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns
3
The isfinite macro returns a nonzero value if and only if its argument has a finite value.

7.12.3.4 The isinf macro
Synopsis
1

#include <math.h>
int isinf(real-floating x);

Description
2
The isinf macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

247) Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal long double value might become subnormal when converted to double, and zero when converted to float.
Returns
The `isinf` macro returns a nonzero value if and only if its argument has an infinite value.

7.12.3.5 The `isnan` macro
Synopsis
```
#include <math.h>
int isnan(real-floating x);
```

Description
The `isnan` macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.\footnote{The `isnan` macro, the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.}

Returns
The `isnan` macro returns a nonzero value if and only if its argument has a NaN value.

7.12.3.6 The `isnormal` macro
Synopsis
```
#include <math.h>
int isnormal(real-floating x);
```

Description
The `isnormal` macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns
The `isnormal` macro returns a nonzero value if and only if its argument has a normal value.

7.12.3.7 The `signbit` macro
Synopsis
```
#include <math.h>
int signbit(real-floating x);
```

Description
The `signbit` macro determines whether the sign of its argument value is negative.\footnote{The `signbit` macro reports the sign of all values, including infinities, zeros, and NaNs. If zero is unsigned, it is treated as positive.}

Returns
The `signbit` macro returns a nonzero value if and only if the sign of its argument value is negative.

7.12.3.8 The `issignaling` macro
Synopsis
```
#include <math.h>
int issignaling(real-floating x);
```

Description
The `issignaling` macro determines whether its argument value is a signaling NaN.

Returns
The **issignaling** macro returns a nonzero value if and only if its argument is a signaling NaN.\(^{250}\)

### 7.12.3.9 The **issubnormal** macro

**Synopsis**

```c
#include <math.h>
int issubnormal(real-floating x);
```

**Description**

The **issubnormal** macro determines whether its argument value is subnormal. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

**Returns**

The **issubnormal** macro returns a nonzero value if and only if its argument is subnormal.

### 7.12.3.10 The **iszero** macro

**Synopsis**

```c
#include <math.h>
int iszero(real-floating x);
```

**Description**

The **iszero** macro determines whether its argument value is (positive, negative, or unsigned) zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then, determination is based on the type of the argument.

**Returns**

The **iszero** macro returns a nonzero value if and only if its argument is zero.

### 7.12.4 Trigonometric functions

#### 7.12.4.1 The **acos** functions

**Synopsis**

```c
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);

#if defined.'<STDC_IEC_60559-DFP-
._Decimal32 acosd32(_Decimal32 x);
._Decimal64 acosd64(_Decimal64 x);
._Decimal128 acosd128(_Decimal128 x);
#endif
```

**Description**

The **acos** functions compute the principal value of the arc cosine of \(x\). A domain error occurs for arguments not in the interval \([-1, +1]\).

**Returns**

The **acos** functions return \(\arccos x\) in the interval \([0, \pi]\) radians.

---

\(^{250}\) F.3 specifies that **issignaling** (and all the other classification macros), raise no floating-point exception if the argument is a variable, or any other expression whose value is represented in the format of its semantic type, even if the value is a signaling NaN.
7.12.4.2 The \texttt{asin} functions

\begin{verbatim}
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 asind32(_Decimal32 x);
  _Decimal64 asind64(_Decimal64 x);
  _Decimal128 asind128(_Decimal128 x);
#endif
\end{verbatim}

\section*{Description}

The \texttt{asin} functions compute the principal value of the arc sine of \(x\). A domain error occurs for arguments not in the interval \([-1, +1]\).

\section*{Returns}

The \texttt{asin} functions return \(\arcsin x\) in the interval \([-\pi/2, +\pi/2]\) \([\pi/2, +\pi/2]\) radians.

7.12.4.3 The \texttt{atan} functions

\begin{verbatim}
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 atand32(_Decimal32 x);
  _Decimal64 atand64(_Decimal64 x);
  _Decimal128 atand128(_Decimal128 x);
#endif
\end{verbatim}

\section*{Description}

The \texttt{atan} functions compute the principal value of the arc tangent of \(x\).

\section*{Returns}

The \texttt{atan} functions return \(\arctan x\) in the interval \([-\pi/2, +\pi/2]\) \([-\pi/2, +\pi/2]\) radians.

7.12.4.4 The \texttt{atan2} functions

\begin{verbatim}
#include <math.h>

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 atan2d32(_Decimal32 y, _Decimal32 x);
  _Decimal64 atan2d64(_Decimal64 y, _Decimal64 x);
  _Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);
#endif
\end{verbatim}

\section*{Description}

The \texttt{atan2} functions compute the value of the arc tangent of \(y/x\), using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

\section*{Returns}

The \texttt{atan2} functions return \(\arctan y/x\) in the interval \([-\pi, +\pi]\) radians.
7.12.4.5 The cos functions

Synopsis

```c
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 cosd32(_Decimal32 x);
  _Decimal64 cosd64(_Decimal64 x);
  _Decimal128 cosd128(_Decimal128 x);
#endif
```

Description

The cos functions compute the cosine of $x$ (measured in radians).

Returns

The cos functions return $\cos x$.

7.12.4.6 The sin functions

Synopsis

```c
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
#ifdef __STDC_IEC_60599_DFP__
  _Decimal32 sind32(_Decimal32 x);
  _Decimal64 sind64(_Decimal64 x);
  _Decimal128 sind128(_Decimal128 x);
#endif
```

Description

The sin functions compute the sine of $x$ (measured in radians).

Returns

The sin functions return $\sin x$.

7.12.4.7 The tan functions

Synopsis

```c
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(long double x);
#ifdef __STDC_IEC_60599_DFP__
  _Decimal32 tand32(_Decimal32 x);
  _Decimal64 tand64(_Decimal64 x);
  _Decimal128 tand128(_Decimal128 x);
#endif
```

Description

The tan functions return the tangent of $x$ (measured in radians).

Returns

The tan functions return $\tan x$.

7.12.4.8 The acospi functions

Synopsis

```c
```

204 Library § 7.12.4.8
### 7.12.4.9 The \( \text{asinpi} \) functions

#### Synopsis

```c
#include <math.h>

double asinpi(double x);
float asinpf(float x);
long double asinpl(long double x);

// Additional functions for Decimal types
#define _STD*math.h
#include <math.h>

__Decimal32 asinpid32(__Decimal32 x);
__Decimal64 asinpid64(__Decimal64 x);
__Decimal128 asinpid128(__Decimal128 x);
#endif
```

#### Description

2 The \( \text{asinpi} \) functions compute the principal value of the arc sine of \( x \), divided by \( \pi \), thus measuring the angle in half-revolutions. A range error occurs if the magnitude of nonzero \( x \) is too small.

#### Returns

3 The \( \text{asinpi} \) functions return \( \text{arc sine}(x)/\pi \) in the interval \([-1, +1] \).

### 7.12.4.10 The \( \text{atanpi} \) functions

#### Synopsis

```c
#include <math.h>

double atanpi(double x);
float atanpif(float x);
long double atanpl(long double x);

// Additional functions for Decimal types
#define _STD*math.h
#include <math.h>

__Decimal32 atanpid32(__Decimal32 x);
__Decimal64 atanpid64(__Decimal64 x);
__Decimal128 atanpid128(__Decimal128 x);
#endif
```

#### Description

2 The \( \text{atanpi} \) functions compute the principal value of the arc tangent of \( x \), divided by \( \pi \), thus measuring the angle in half-revolutions. A range error occurs if the magnitude of nonzero \( x \) is too small.

#### Returns

3 The \( \text{atanpi} \) functions return \( \text{arc tangent}(x)/\pi \) in the interval \([-1, +1] \).
7.12.4.11 The atan2pi functions

Synopsis

```c
#include <math.h>
double atan2pi(double y, double x);
float atan2pif(float y, float x);
long double atan2pil(long double y, long double x);
#ifdef __STDC_IEC_60559_DFP__
_decimal32 atan2pid32(_Decimal32 y, _Decimal32 x);
_decimal64 atan2pid64(_Decimal64 y, _Decimal64 x);
_decimal128 atan2pid128(_Decimal128 y, _Decimal128 x);
#endif
```

Description

2 The atan2pi functions compute the angle, measured in half-revolutions, subtended at the origin by the point \((x, y)\) and the positive \(x\)-axis. Thus, the atan2pi functions compute \(\arctan(\frac{y}{x})/\pi\) in the range \([-1, +1]\). A domain error may occur if both arguments are zero. A range error occurs if \(x\) is positive and the magnitude of nonzero \(\frac{y}{x}\) is too small.

Returns

3 The atan2pi functions return the computed angle, in the interval \([-1, +1]\).

7.12.4.12 The cospi functions

Synopsis

```c
#include <math.h>
double cospi(double x);
float cospif(float x);
long double cospil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_decimal32 cospid32(_Decimal32 x);
_decimal64 cospid64(_Decimal64 x);
_decimal128 cospid128(_Decimal128 x);
#endif
```

Description

2 The cospi functions compute the cosine of \(\pi \times x\), thus regarding \(x\) as a measurement in half-revolutions.

Returns

3 The cospi functions return \(\cos(\pi \times x)\).

7.12.4.13 The sinpi functions

Synopsis

```c
#include <math.h>
double sinpi(double x);
float sinpif(float x);
long double sinpil(long double x);
#ifdef __STDC_IEC_60559_DFP__
_decimal32 sinpid32(_Decimal32 x);
_decimal64 sinpid64(_Decimal64 x);
_decimal128 sinpid128(_Decimal128 x);
#endif
```

Description

2 The sinpi functions compute the sine of \(\pi \times x\), thus regarding \(x\) as a measurement in half-revolutions.

Returns
The \texttt{sinpi} functions return \( \sin(\pi \times x) \).

\subsection*{7.12.4.14 The \texttt{tanpi} functions}

\textbf{Synopsis}

```c
#include <math.h>
double tanpi(double x);
float tanpif(float x);
long double tanpil(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 tanpidual32(_Decimal32 x);
  _Decimal64 tanpidual64(_Decimal64 x);
  _Decimal128 tanpidual128(_Decimal128 x);
#endif
```

\textbf{Description}

The \texttt{tanpi} functions compute the tangent of \( \pi \times x \), thus regarding \( x \) as a measurement in half-revolutions.

\textbf{Returns}

The \texttt{tanpi} functions return \( \tan(\pi \times x) \).

\subsection*{7.12.5 Hyperbolic functions}

\subsection*{7.12.5.1 The \texttt{acosh} functions}

\textbf{Synopsis}

```c
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 acoshd32(_Decimal32 x);
  _Decimal64 acoshd64(_Decimal64 x);
  _Decimal128 acoshd128(_Decimal128 x);
#endif
```

\textbf{Description}

The \texttt{acosh} functions compute the (nonnegative) arc hyperbolic cosine of \( x \). A domain error occurs for arguments less than 1.

\textbf{Returns}

The \texttt{acosh} functions return \( \text{arcosh} \ x \) in the interval \([0, +\infty]\).

\subsection*{7.12.5.2 The \texttt{asinh} functions}

\textbf{Synopsis}

```c
#include <math.h>
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 asinhd32(_Decimal32 x);
  _Decimal64 asinhd64(_Decimal64 x);
  _Decimal128 asinhd128(_Decimal128 x);
#endif
```

\textbf{Description}

The \texttt{asinh} functions compute the arc hyperbolic sine of \( x \).
Returns

3 The **asinh** functions return arsinh \( x \).

### 7.12.5.3 The **atanh** functions

#### Synopsis

```
#include <math.h>

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);

#ifdef __STDC_IEC_60559_DFP__

Decimal32 tanhd32(Decimal32 x);
Decimal64 tanhd64(Decimal64 x);
Decimal128 tanhd128(Decimal128 x);
#endif
```

**Description**

2 The **atanh** functions compute the arc hyperbolic tangent of \( x \). A domain error occurs for arguments not in the interval \([-1, +1]\). A pole error may occur if the argument equals -1 or +1.

Returns

3 The **atanh** functions return artanh \( x \).

### 7.12.5.4 The **cosh** functions

#### Synopsis

```
#include <math.h>

double cosh(double x);
float coshf(float x);
long double coshl(long double x);

#ifdef __STDC_IEC_60559_DFP__

Decimal32 coshd32(Decimal32 x);
Decimal64 coshd64(Decimal64 x);
Decimal128 coshd128(Decimal128 x);
#endif
```

**Description**

2 The **cosh** functions compute the hyperbolic cosine of \( x \). A range error occurs if the magnitude of \( x \) is too large.

Returns

3 The **cosh** functions return cosh \( x \).

### 7.12.5.5 The **sinh** functions

#### Synopsis

```
#include <math.h>

double sinh(double x);
float sinhf(float x);
long double sinhhl(long double x);

#ifdef __STDC_IEC_60559_DFP__

Decimal32 sinhld32(Decimal32 x);
Decimal64 sinhld64(Decimal64 x);
Decimal128 sinhld128(Decimal128 x);
#endif
```

**Description**

2 The **sinh** functions compute the hyperbolic sine of \( x \). A range error occurs if the magnitude of \( x \) is too large.
Returns
3 The **sinh** functions return **sinh** \( x \).

7.12.5.6 The **tanh** functions

**Synopsis**
```c
#include <math.h>

double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
#endif
__STDC_IEC_60559_DFP__
_Decimal32 tanhd32(_Decimal32 x);
_Decimal64 tanhd64(_Decimal64 x);
_Decimal128 tanhd128(_Decimal128 x);
#endif
```

**Description**
2 The **tanh** functions compute the hyperbolic tangent of \( x \).

Returns
3 The **tanh** functions return **tanh** \( x \).

7.12.6 Exponential and logarithmic functions

7.12.6.1 The **exp** functions

**Synopsis**
```c
#include <math.h>

double exp(double x);
float expf(float x);
long double expl(long double x);
#endif
__STDC_IEC_60559_DFP__
_Decimal32 expd32(_Decimal32 x);
_Decimal64 expd64(_Decimal64 x);
_Decimal128 expd128(_Decimal128 x);
#endif
```

**Description**
2 The **exp** functions compute the base-\( e \) exponential of \( x \). A range error occurs if the magnitude of \( x \) is too large.

Returns
3 The **exp** functions return \( e^x \).

7.12.6.2 The **exp10** functions

**Synopsis**
```c
#include <math.h>

double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
#endif
__STDC_IEC_60559_DFP__
_Decimal32 exp10d32(_Decimal32 x);
_Decimal64 exp10d64(_Decimal64 x);
_Decimal128 exp10d128(_Decimal128 x);
#endif
```

**Description**
2 The **exp10** functions compute the base-10 exponential of \( x \). A range error occurs if the magnitude of finite \( x \) is too large.
Returns

The `exp10` functions return $10^x$.

### The `exp10` functions

**Synopsis**

```c
#include <math.h>

double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);

#ifdef __STDC_IEC_60559_DFP__

_Decimal32 exp10m1d32(_Decimal32 x);
_Decimal64 exp10m1d64(_Decimal64 x);
_Decimal128 exp10m1d128(_Decimal128 x);
#endif
```

**Description**

The `exp10` functions compute the base-10 exponential of the argument, minus 1. A range error occurs if finite `x` is too large or if the magnitude of nonzero `x` is too small.

Returns

The `exp10` functions return $10^x - 1$.

### The `exp2` functions

**Synopsis**

```c
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

#ifdef __STDC_IEC_60559_DFP__

_Decimal32 exp2d32(_Decimal32 x);
_Decimal64 exp2d64(_Decimal64 x);
_Decimal128 exp2d128(_Decimal128 x);
#endif
```

**Description**

The `exp2` functions compute the base-2 exponential of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

The `exp2` functions return $2^x$.

### The `exp2m1` functions

**Synopsis**

```c
#include <math.h>

double exp2m1(double x);
float exp2mf(float x);
long double exp2ml(long double x);

#ifdef __STDC_IEC_60559_DFP__

_Decimal32 exp2m1d32(_Decimal32 x);
_Decimal64 exp2m1d64(_Decimal64 x);
_Decimal128 exp2m1d128(_Decimal128 x);
#endif
```

**Description**

The `exp2m1` functions compute the base-2 exponential of the argument, minus 1. A range error occurs if the magnitude of `x` is too large or if the magnitude of nonzero `x` is too small.
7.12.6.6 The **expm1** functions

**Synopsis**

```c
#include <math.h>

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

**Description**

The **expm1** functions compute the base-\(e\) exponential of the argument, minus 1. A range error occurs if positive \(x\) is too large.\(^{251}\)

**Returns**

The **expm1** functions return \(e^x - 1\).

7.12.6.7 The **frexp** functions

**Synopsis**

```c
#include <math.h>

double frexp(double value, int *p);
float frexpf(float value, int *p);
long double frexpl(long double value, int *p);
```

**Description**

The **frexp** functions break a floating-point number into a normalized fraction and an integral power of 2 (integer exponent). They store the integer in the **int** object pointed to by **exp**. If the type of the function is a standard floating type, the exponent is an integral power of 2. If the type of the function is a decimal floating type, the exponent is an integral power of 10.

**Returns**

If **value** is not a floating-point number or if the integral power of 2 is outside the range of **int**, the results are unspecified. Otherwise, the **frexp** functions return the value \(x\), such that \(x\) has a magnitude in the interval \([\frac{1}{2}, 1)\) or zero, and value equals \(x \times 2^e\), when the type of the function is a standard floating type; or \(x\) has a magnitude in the interval \([1/10, 1)\) or zero, and value equals \(x \times 10^e\), when the type of the function is a decimal floating type. If **value** is zero, both parts of the result are zero.

7.12.6.8 The **ilogb** functions

**Synopsis**

```c
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

\(^{251}\)For small magnitude \(x\), **expm1**(\(x\)) is expected to be more accurate than **exp**(\(x\)) - 1.
### Synopsis replace `Synopsis`

#### 7.12.6.9 The `ldexp` functions

```c
#include <math.h>

double ldexp(double x, int p);
float ldexpf(float x, int p);
long double ldexp(long double x, int p);
#endif
```

**Description**

2 The `ldexp` functions multiply a floating-point number by an integral power of 2. When the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type. A range error may occur.

**Returns**

3 The `ldexp` functions return $x \times 2^p$ when the type of the function is a standard floating type, or return $x \times 10^p$ when the type of the function is a decimal floating type.

#### 7.12.6.10 The `llogb` functions

**Synopsis**

```c
#include <math.h>

long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
#endif
```

**Description**

2 The `llogb` functions extract the exponent of $x$ as a signed `long int` value. If $x$ is zero they compute the value `FP_ILLOGB0`; if $x$ is infinite they compute the value `LONG_MAX`; if $x$ is a NaN they compute the value `FP_ILLOGBNAN`; otherwise, they are equivalent to calling the corresponding `logb` function and converting the returned value to type `long int`. A domain error or range error may occur if

```
int ilogbd32(_Decimal32 x);
int ilogbd64(_Decimal64 x);
int ilogbd128(_Decimal128 x);
#endif
```
x is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified.

Returns

3 The llogb functions return the exponent of x as a signed long int value.

Forward references: the logb functions (7.12.6.17).

7.12.6.11 The log functions

Synopsis replace Synopsis

```c
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 logd32(_Decimal32 x);
_Decimal64 logd64(_Decimal64 x);
_Decimal128 logd128(_Decimal128 x);
#endif
```

Description

2 The log functions compute the base-e (natural) logarithm of x. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The log functions return log_e x.

7.12.6.12 The log10 functions

Synopsis replace Synopsis

```c
#include <math.h>
double log10(double x);
float log10f(float x);
long double log10l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log10d32(_Decimal32 x);
_Decimal64 log10d64(_Decimal64 x);
_Decimal128 log10d128(_Decimal128 x);
#endif
```

Description

2 The log10 functions compute the base-10 (common) logarithm of x. A domain error occurs if the argument is negative. A pole error may occur if the argument is zero.

Returns

3 The log10 functions return log_10 x.

7.12.6.13 The log10p1 functions

Synopsis replace Synopsis

```c
#include <math.h>
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 log10p1d32(_Decimal32 x);
_Decimal64 log10p1d64(_Decimal64 x);
_Decimal128 log10p1d128(_Decimal128 x);
#endif
```
Description

The \texttt{\texttt{log10p1}} functions compute the base-10 logarithm of 1 plus the argument. A domain error occurs if the argument is less than \(-1\). A pole error may occur if the argument equals \(-1\). A range error occurs if the magnitude of nonzero \(x\) is too small.

Returns

The \texttt{\texttt{log10p1}} functions return \(\log_{10}(1 + x)\).

\section*{Synopsis}

\begin{verbatim}
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
double logp1(double x);
float logp1f(float x);
long double logp1l(long double x);
#endif
#endif
\end{verbatim}

\section*{Description}

The \texttt{\texttt{log1p}} functions are equivalent to the \texttt{\texttt{logp1}} functions. These functions compute the base-\(e\) (natural) logarithm of 1 plus the argument.\(^{252}\) A domain error occurs if the argument is less than \(-1\). A pole error may occur if the argument equals \(-1\).

Returns

The \texttt{\texttt{log1p}} and \texttt{\texttt{logp1}} functions return \(\log_e(1 + x)\).

\section*{Synopsis}

\begin{verbatim}
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
#endif
#endif
\end{verbatim}

\section*{Description}

The \texttt{\texttt{log2}} functions compute the base-2 logarithm of \(x\). A domain error occurs if the argument is less than zero. A pole error may occur if the argument is zero.

Returns

The \texttt{\texttt{log2}} functions return \(\log_2 x\).

\section*{Synopsis}

\begin{verbatim}
#include <math.h>
double log2p1(double x);
float log2pf(float x);
long double log2pl(long double x);
#endif
#endif
\end{verbatim}

\section*{Description}

For small magnitude \(x\), \(\texttt{\texttt{logp1}}(x)\) is expected to be more accurate than \(\log(1 + x)\).

\section*{Library}
The \texttt{log2p1} functions compute the base-2 logarithm of 1 plus the argument. A domain error occurs if the argument is less than $-1$. A pole error may occur if the argument equals $-1$.

The \texttt{log2p1} functions return $\log_2(1+x)$.

The \texttt{logb} functions extract the exponent of $x$, as a signed integer value in floating-point format. If $x$ is subnormal it is treated as though it were normalized; thus, for positive finite $x$,

$$1 \leq x \times \text{FLT\_RADX}^b \log_b(x) < \text{FLT\_RADX}^b$$

where $b = \text{FLT\_RADX}$ if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type. A domain error or pole error may occur if the argument is zero.

The \texttt{logb} functions return the signed exponent of $x$.

The \texttt{modf} functions break the argument \texttt{value} into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in floating-point format) in the object pointed to by \texttt{iptr}.
Returns
3 The **modf** functions return the signed fractional part of **value**.

### 7.12.6.19 The **scalbn** and **scalbln** functions

#### Synopsis

```c
#include <math.h>

double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbln(long double x, long int n);
float scalblnf(float x, long int n);

#define __STDC_IEC_60559_DFP__

_decimal32 scalbnd32(_Decimal32 x, int n);
_decimal64 scalbnd64(_Decimal64 x, int n);
_decimal128 scalbnd128(_Decimal128 x, int n);
_decimal32 scalblnd32(_Decimal32 x, long int n);
_decimal64 scalblnd64(_Decimal64 x, long int n);
_decimal128 scalblnd128(_Decimal128 x, long int n);
#endif
```

#### Description
2 The **scalbn** and **scalbln** functions compute \( x \times \text{FLT}_n \) efficiently, not normally by computing \( \text{FLT}_n \) explicitly, \( x \times b^n \), where \( b = \text{FLT}_n \) if the type of the function is a standard floating type, or \( b = 10 \) if the type of the function is a decimal floating type. A range error may occur.

#### Returns
3 The **scalbn** and **scalbln** functions return \( x \times \text{FLT}_n \).

### 7.12.7 Power and absolute-value functions

#### 7.12.7.1 The **cbrt** functions

#### Synopsis

```c
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

#define __STDC_IEC_60559_DFP__

_decimal32 cbrtd32(_Decimal32 x);
_decimal64 cbrtd64(_Decimal64 x);
_decimal128 cbrtd128(_Decimal128 x);
#endif
```

#### Description
2 The **cbrt** functions compute the real cube root of **x**.

#### Returns
3 The **cbrt** functions return \( x^{\frac{1}{3}} \).

#### 7.12.7.2 The **compoundn** functions

#### Synopsis

```c
#include <stdint.h>
#include <math.h>

double compoundn(double x, intmax_t n);
float compoundnf(float x, intmax_t n);
long double compoundnl(long double x, intmax_t n);

#define __STDC_IEC_60559_DFP__

_decimal32 compoundnd32(_Decimal32 x, intmax_t n);
_decimal64 compoundnd64(_Decimal64 x, intmax_t n);
_decimal128 compoundnd128(_Decimal128 x, intmax_t n);
```

#### Description
2 The **compoundn** functions compute \( \text{A} \times x^\text{B} \) efficiently, not normally by computing \( \text{A} \times x^\text{B} \) explicitly.
The `compoundn` functions compute 1 plus \( x \), raised to the power \( n \). A domain error occurs if \( x \lt -1 \). A range error may occur if \( n \) is too large, depending on \( x \). A pole error may occur if \( x \) equals \(-1\) and \( n \lt 0 \).

**Returns**

The `compoundn` functions return \((1 + x)^n\).

### 7.12.7.3 The `fabs` functions

**Synopsis**

```c
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
#endif
```

**Description**

The `fabs` functions compute the absolute value of a floating-point number \( x \).

**Returns**

The `fabs` functions return \(|x|\).

### 7.12.7.4 The `hypot` functions

**Synopsis**

```c
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
#endif
```

**Description**

The `hypot` functions compute the square root of the sum of the squares of \( x \) and \( y \), without undue overflow or underflow. A range error may occur.

**Returns**

The `hypot` functions return \(\sqrt{x^2 + y^2}\).

### 7.12.7.5 The `pow` functions

**Synopsis**

```c
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

**Description**

The `pow` functions
Description
The \texttt{pow} functions compute $x$ raised to the power $y$. A domain error occurs if $x$ is finite and negative and $y$ is finite and not an integer value. A range error may occur. A domain error may occur if $x$ is zero and $y$ is zero. A domain error or pole error may occur if $x$ is zero and $y$ is less than zero.

Returns
The \texttt{pow} functions return $x^y$.

7.12.7.6 The \texttt{pown} functions

Synopsis

```c
#include <stdint.h>
#include <math.h>
double pown(double x, intmax_t n);
float pownf(float x, intmax_t n);
long double pownl(long double x, intmax_t n);
#endif

_DECIMAL32 pown32(_Decimal32 x, _Decimal32 y);
_DECIMAL64 pown64(_Decimal64 x, _Decimal64 y);
_DECIMAL128 pown128(_Decimal128 x, _Decimal128 y);
#endif
```

Description
The \texttt{pown} functions compute $x$ raised to the $n$th power. A range error may occur. A pole error may occur if $x$ equals 0 and $n < 0$.

Returns
The \texttt{pown} functions return $x^n$.

7.12.7.7 The \texttt{powr} functions

Synopsis

```c
#include <math.h>
double powr(double y, double x);
float powrf(float y, float x);
long double powrl(long double y, long double x);
#endif

_DECIMAL32 powr32(_Decimal32 y, _Decimal32 x);
_DECIMAL64 powr64(_Decimal64 y, _Decimal64 x);
_DECIMAL128 powr128(_Decimal128 y, _Decimal128 x);
#endif
```

Description
The \texttt{powr} functions compute $x$ raised to the power $y$ as $e^{y \log x}$. A domain error occurs if $x < 0$ or if $x$ and $y$ are both zero. A range error may occur. A pole error may occur if $x$ equals zero and finite $y < 0$.

Returns
The \texttt{powr} functions return $x^y$.

7.12.7.8 The \texttt{rotn} functions

Synopsis
Description
2 The \texttt{rootn} functions compute the principal $n$th root of $x$. A domain error occurs if $n$ is 0 or if $x < 0$ and $n$ is even. A range error may occur if $n$ is $-1$. A pole error may occur if $x$ equals zero and $n < 0$.

Returns
3 The \texttt{rootn} functions return $x^{\frac{1}{n}}$.

7.12.7.9 The \texttt{rsqrt} functions

Synopsis
1
```c
#include <math.h>
double rsqrt(double x);
float rsqrtf(float x);
long double rsqrtl(long double x);
#endif
`_Decimal32 rsqrtd32(_Decimal32 x, intmax_t n);
`_Decimal64 rsqrtd64(_Decimal64 x, intmax_t n);
`_Decimal128 rsqrtd128(_Decimal128 x, intmax_t n);
#endif
```

Description
2 The \texttt{rsqrt} functions compute the reciprocal of the square root of the argument. A domain error occurs if the argument is less than zero. A pole error may occur if the argument equals zero.

Returns
3 The \texttt{rsqrt} functions return $\frac{1}{\sqrt{x}}$.

7.12.7.10 The \texttt{sqrt} functions

Synopsis
1
```c
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
#endif
`_Decimal32 sqrtd32(_Decimal32 x);
`_Decimal64 sqrtd64(_Decimal64 x);
`_Decimal128 sqrtd128(_Decimal128 x);
#endif
```

Description
2 The \texttt{sqrt} functions compute the nonnegative square root of $x$. A domain error occurs if the argument is less than zero.

Returns
3 The \texttt{sqrt} functions return $\sqrt{x}$.
7.12.8 Error and gamma functions

7.12.8.1 The erf functions

**Synopsis**
```c
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 erfd32(_Decimal32 x);
  _Decimal64 erfd64(_Decimal64 x);
  _Decimal128 erfd128(_Decimal128 x);
#endif
```

**Description**

The `erf` functions compute the error function of `x`.

**Returns**

The `erf` functions return \( \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt \).

7.12.8.2 The erfc functions

**Synopsis**
```c
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcfl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 erfcfd32(_Decimal32 x);
  _Decimal64 erfcfd64(_Decimal64 x);
  _Decimal128 erfcfd128(_Decimal128 x);
#endif
```

**Description**

The `erfc` functions compute the complementary error function of `x`. A range error occurs if positive `x` is too large.

**Returns**

The `erfc` functions return \( \text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt \).

7.12.8.3 The lgamma functions

**Synopsis**
```c
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 lgammad32(_Decimal32 x);
  _Decimal64 lgammad64(_Decimal64 x);
  _Decimal128 lgammad128(_Decimal128 x);
#endif
```

**Description**

The `lgamma` functions compute the natural logarithm of the absolute value of gamma of `x`. A range error occurs if positive `x` is too large. A pole error may occur if `x` is a negative integer or zero.
Returns
3 The \texttt{lgamma} functions return $\log_e |\Gamma(x)|$.

7.12.8.4 The \texttt{tgamma} functions

Synopsis

```c
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 tgammad32(_Decimal32 x);
  _Decimal64 tgammad64(_Decimal64 x);
  _Decimal128 tgammad128(_Decimal128 x);
#endif
```

Description
2 The \texttt{tgamma} functions compute the gamma function of $x$. A domain error or pole error may occur if $x$ is a negative integer or zero. A range error occurs if the magnitude of $x$ is too large and may occur if the magnitude of $x$ is too small.

Returns
3 The \texttt{tgamma} functions return $\Gamma(x)$.

7.12.9 Nearest integer functions

7.12.9.1 The \texttt{ceil} functions

Synopsis

```c
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 ceild32(_Decimal32 x);
  _Decimal64 ceild64(_Decimal64 x);
  _Decimal128 ceild128(_Decimal128 x);
#endif
```

Description
2 The \texttt{ceil} functions compute the smallest integer value not less than $x$.

Returns
3 The \texttt{ceil} functions return $\lceil x \rceil$, expressed as a floating-point number.

7.12.9.2 The \texttt{floor} functions

Synopsis

```c
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 floord32(_Decimal32 x);
  _Decimal64 floord64(_Decimal64 x);
  _Decimal128 floord128(_Decimal128 x);
#endif
```

Description
2 The \texttt{floor} functions compute the largest integer value not greater than $x$.

§ 7.12.9.2 Library 221
Returns
3 The \texttt{floor} functions return $\lfloor x \rfloor$, expressed as a floating-point number.

7.12.9.3 The \texttt{nearbyint} functions

Synopsis
1
\begin{verbatim}
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal32 nearbyintd32(_Decimal32 x);
 _Decimal64 nearbyintd64(_Decimal64 x);
 _Decimal128 nearbyintd128(_Decimal128 x);
#endif
\end{verbatim}

Description
2 The \texttt{nearbyint} functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the “inexact” floating-point exception.

Returns
3 The \texttt{nearbyint} functions return the rounded integer value.

7.12.9.4 The \texttt{rint} functions

Synopsis
1
\begin{verbatim}
#include <math.h>

double rint(double x);
float rintf(float x);
long double rintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal32 rintd32(_Decimal32 x);
 _Decimal64 rintd64(_Decimal64 x);
 _Decimal128 rintd128(_Decimal128 x);
#endif
\end{verbatim}

Description
2 The \texttt{rint} functions differ from the \texttt{nearbyint} functions (7.12.9.3) only in that the \texttt{rint} functions may raise the “inexact” floating-point exception if the result differs in value from the argument.

Returns
3 The \texttt{rint} functions return the rounded integer value.

7.12.9.5 The \texttt{lrint} and \texttt{llrint} functions

Synopsis
1
\begin{verbatim}
#include <math.h>

long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
#ifdef __STDC_IEC_60559_DFP__
 long int lrintd32(_Decimal32 x);
 long int lrintd64(_Decimal64 x);
 long int lrintd128(_Decimal128 x);
#endif
\end{verbatim}

222 Library § 7.12.9.5
Description
2 The `lrint` and `llrint` functions round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns
3 The `lrint` and `llrint` functions return the rounded integer value.

7.12.9.6 The round functions

```c
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
#if defined __STDC_IEC_60559_DFP__
  _Decimal32 roundd32(_Decimal32 x);
  _Decimal64 roundd64(_Decimal64 x);
  _Decimal128 roundd128(_Decimal128 x);
#endif
```

Description
2 The `round` functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Returns
3 The `round` functions return the rounded integer value.

7.12.9.7 The lround and llround functions

```c
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
#if defined __STDC_IEC_60559_DFP__
  long int lroundd32(_Decimal32 x);
  long int lroundd64(_Decimal64 x);
  long int lroundd128(_Decimal128 x);
  long long int llroundd32(_Decimal32 x);
  long long int llroundd64(_Decimal64 x);
  long long int llroundd128(_Decimal128 x);
#endif
```

Description
2 The `lround` and `llround` functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns
3 The `lround` and `llround` functions return the rounded integer value.

7.12.9.8 The roundeven functions

```c
#include <math.h>
```

§ 7.12.9.8 Library 223
double roundeven(double x);
float roundevenf(float x);
long double roundevenl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 roundevend32(_Decimal32 x);
  _Decimal64 roundevend64(_Decimal64 x);
  _Decimal128 roundevend128(_Decimal128 x);
#endif

Description

2. The `roundeven` functions round their argument to the nearest integer value in floating-point format, rounding midway cases to even (that is, to the nearest value that is an even integer), regardless of the current rounding direction.

Returns

3. The `roundeven` functions return the rounded integer value.

7.12.9.9 The `trunc` functions

Synopsis

```c
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 truncd32(_Decimal32 x);
  _Decimal64 truncd64(_Decimal64 x);
  _Decimal128 truncd128(_Decimal128 x);
#endif
```

Description

2. The `trunc` functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

3. The `trunc` functions return the truncated integer value.

7.12.9.10 The `fromfp` and `ufromfp` functions

Synopsis

```c
#include <stdint.h>
#include <math.h>

intmax_t fromfp(double x, int round, unsigned int width);
intmax_t fromfpf(float x, int round, unsigned int width);
intmax_t fromfpl(long double x, int round, unsigned int width);
uintmax_t ufromfp(double x, int round, unsigned int width);
uintmax_t ufromfpf(float x, int round, unsigned int width);
uintmax_t ufromfpl(long double x, int round, unsigned int width);
#ifdef __STDC_IEC_60559_DFP__
  intmax_t fromfpd32(_Decimal32 x, int round, unsigned int width);
  intmax_t fromfpd64(_Decimal64 x, int round, unsigned int width);
  intmax_t fromfpd128(_Decimal128 x, int round, unsigned int width);
  uintmax_t ufromfpd32(_Decimal32 x, int round, unsigned int width);
  uintmax_t ufromfpd64(_Decimal64 x, int round, unsigned int width);
  uintmax_t ufromfpd128(_Decimal128 x, int round, unsigned int width);
#endif
```

Description

2. The `fromfp` and `ufromfp` functions round `x`, using the math rounding direction indicated by `round`, to a signed or unsigned integer, respectively, of `width` bits, and return the result value.
in the integer type designated by \texttt{intmax\_t} or \texttt{uintmax\_t}, respectively. If the value of the round argument is not equal to the value of a math rounding direction macro, the direction of rounding is unspecified. If the value of width exceeds the width of the function type, the rounding is to the full width of the function type. The \texttt{fmod} and \texttt{ufromfp} functions do not raise the “inexact” floating-point exception. If \( x \) is infinite or \( \text{NaN} \) or rounds to an integral value that is outside the range of any supported integer type\footnote{For signed types, 6.2.6.2 permits three representations, which differ in whether a value of \(-(2^M)\), where \( M \) is the number of value bits, can be represented.} of the specified width, or if width is zero, the functions return an unspecified value and a domain error occurs.

**Returns**

3 The \texttt{fmod} and \texttt{ufromfp} functions return the rounded integer value.

EXEMPLARY Upward rounding of \texttt{double} \( x \) to type \texttt{int}, without raising the “inexact” floating-point exception, is achieved by

\[
\text{(int)fmod}(x, \text{FP\_INT\_UPWARD, INT\_WIDTH})
\]

### 7.12.9.11 The \texttt{fromfp} and \texttt{ufromfp} functions

**Synopsis**

1

\begin{verbatim}
#include <stdint.h>
#include <math.h>

intmax_t fromfp(double x, int round, unsigned int width);
intmax_t fromfpf(float x, int round, unsigned int width);
intmax_t fromfpfl(long double x, int round, unsigned int width);
uintmax_t ufromfp(double x, int round, unsigned int width);
uintmax_t ufromfpf(float x, int round, unsigned int width);
uintmax_t ufromfpfl(long double x, int round, unsigned int width);

#ifdef __STDC_IEC_60559_DFP__
intmax_t fromfpd32(_Decimal32 x, int round, unsigned int width);
intmax_t fromfpd64(_Decimal64 x, int round, unsigned int width);
intmax_t fromfpd128(_Decimal128 x, int round, unsigned int width);
uintmax_t ufromfpd32(_Decimal32 x, int round, unsigned int width);
uintmax_t ufromfpd64(_Decimal64 x, int round, unsigned int width);
uintmax_t ufromfpd128(_Decimal128 x, int round, unsigned int width);
#endif
\end{verbatim}

**Description**

2 The \texttt{fromfp} and \texttt{ufromfp} functions differ from the \texttt{fmod} and \texttt{ufromfp} functions, respectively, only in that the \texttt{fromfp} and \texttt{ufromfp} functions raise the “inexact” floating-point exception if a rounded result not exceeding the specified width differs in value from the argument \( x \).

**Returns**

3 The \texttt{fromfp} and \texttt{ufromfp} functions return the rounded integer value.

NOTE Conversions to integer types that are not required to raise the inexact exception can be done simply by rounding to integral value in floating type and then converting to the target integer type. For example, the conversion of \texttt{long double} \( x \) to \texttt{uint64\_t} using upward rounding, is done by

\[
\text{(uint64\_t)}\text{ceill}(x)
\]

### 7.12.10 Remainder functions

#### 7.12.10.1 The \texttt{fmod} functions

**Synopsis**

1

\begin{verbatim}
#include <math.h>

double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);

#ifdef __STDC_IEC_60559_DFP__

\end{verbatim}

\footnote{For signed types, 6.2.6.2 permits three representations, which differ in whether a value of \(-(2^M)\), where \( M \) is the number of value bits, can be represented.}

\section*{§ 7.12.10.1}
The \texttt{fmod} functions compute the floating-point remainder of \( x/y \).

Returns

The \texttt{fmod} functions return the value \( x - ny \), for some integer \( n \) such that, if \( y \) is nonzero, the result has the same sign as \( x \) and magnitude less than the magnitude of \( y \). If \( y \) is zero, whether a domain error occurs or the \texttt{fmod} functions return zero is implementation-defined.

7.12.10.2 The \texttt{remainder} functions

Synopsis replace Synopsis

```c
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);

#ifndef __STDC_IEC_60559_DFP__

_decimal32 remainderd32(_decimal32 x, _decimal32 y);
_decimal64 remainderd64(_decimal64 x, _decimal64 y);
_decimal128 remainderd128(_decimal128 x, _decimal128 y);
#endif
```

Description

The \texttt{remainder} functions compute the remainder \( x \) REM \( y \) required by IEC 60559.\footnote{When \( y \neq 0 \), the remainder \( r = x \) REM \( y \) is defined regardless of the rounding mode by the mathematical relation \( r = x - ny \), where \( n \) is the integer nearest the exact value of \( \frac{x}{y} \): whenever \( |n - \frac{x}{y}| = \frac{1}{2} \), then \( n \) is even. If \( r = 0 \), its sign shall be that of \( x \). This definition is applicable for all implementations.}

Returns

The \texttt{remainder} functions return \( x \) REM \( y \). If \( y \) is zero, whether a domain error occurs or the functions return zero is implementation-defined.

7.12.10.3 The \texttt{remquo} functions

Synopsis replace Synopsis

```c
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

Description

The \texttt{remquo} functions compute the same remainder as the \texttt{remainder} functions. In the object pointed to by \texttt{quo} they store a value whose sign is the sign of \( x/y \) and whose magnitude is congruent modulo \( 2^n \) to the magnitude of the integral quotient of \( x/y \), where \( n \) is an implementation-defined integer greater than or equal to 3.

Returns

The \texttt{remquo} functions return \( x \) REM \( y \). If \( y \) is zero, the value stored in the object pointed to by \texttt{quo} is unspecified and whether a domain error occurs or the functions return zero is implementation defined.

4 \textbf{NOTE} There are no decimal floating-point versions of the \texttt{remquo} functions. \footnote{When \( y \neq 0 \), the remainder \( r = x \) REM \( y \) is defined regardless of the rounding mode by the mathematical relation \( r = x - ny \), where \( n \) is the integer nearest the exact value of \( \frac{x}{y} \): whenever \( |n - \frac{x}{y}| = \frac{1}{2} \), then \( n \) is even. If \( r = 0 \), its sign shall be that of \( x \). This definition is applicable for all implementations.}

7.12.11 Manipulation functions

7.12.11.1 The \texttt{copysign} functions
Synopsis

1
```c
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 copysign32(_Decimal32 x, _Decimal32 y);
  _Decimal64 copysign64(_Decimal64 x, _Decimal64 y);
  _Decimal128 copysign128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The `copysign` functions produce a value with the magnitude of `x` and the sign of `y`. They produce a NaN (with the sign of `y`) if `x` is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the `copysign` functions regard the sign of zero as positive.

Returns

3 The `copysign` functions return a value with the magnitude of `x` and the sign of `y`.

7.12.11.2 The `nan` functions

Synopsis

1
```c
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 nand32(const char *tagp);
  _Decimal64 nand64(const char *tagp);
  _Decimal128 nand128(const char *tagp);
#endif
```

Description

2 The `nan`, `nanf`, and `nanl` functions convert the string pointed to by `tagp` according to the following rules. The call `nan("n-char-sequence")` is equivalent to `strtod("NAN(n-char-sequence)",(char**)NULL)`. The call `nan("")` is equivalent to `strtod("NAN()",(char**)NULL)`. If `tagp` does not point to an n-char sequence or an empty string, the call is equivalent to `strtod("NAN",(char**)NULL)`. Calls to `nanf` and `nanl` are equivalent to the corresponding calls to `strtof` and `strtold`.

Returns

3 The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the `strtod`, `strtof`, and `strtold` functions (7.22.1.5).

7.12.11.3 The `nextafter` functions

Synopsis

1
```c
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 nextafterd32(_Decimal32 x, _Decimal32 y);
  _Decimal64 nextafterd64(_Decimal64 x, _Decimal64 y);
  _Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);
#endif
```
Description

2 The nextafter functions determine the next representable value, in the type of the function, after \( x \) in the direction of \( y \), where \( x \) and \( y \) are first converted to the type of the function.\(^{255}\) The nextafter functions return \( y \) if \( x \) equals \( y \). A range error may occur if the magnitude of \( x \) is the largest finite value representable in the type and the result is infinite or not representable in the type.

Returns

3 The nextafter functions return the next representable value in the specified format after \( x \) in the direction of \( y \).

7.12.11.4 The nexttoward functions

Synopsis replace Synopsis

```c
#include <math.h>

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

#if defined __STDC_IEC_60559_DFP__
  _Decimal32 nexttoward32(_Decimal32 x, _Decimal128 y);
  _Decimal64 nexttoward64(_Decimal64 x, _Decimal128 y);
  _Decimal128 nexttowardl28(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The nexttoward functions are equivalent to the nextafter functions except that the second parameter has type long double or _Decimal128 and the functions return \( y \) converted to the type of the function if \( x \) equals \( y \).\(^{256}\)

7.12.11.5 The nextup functions

Synopsis

```c
#include <math.h>

double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);

#if defined __STDC_IEC_60559_DFP__
  _Decimal32 nextup32(_Decimal32 x);
  _Decimal64 nextup64(_Decimal64 x);
  _Decimal128 nextupl28(_Decimal128 x);
#endif
```

Description

2 The nextup functions determine the next representable value, in the type of the function, greater than \( x \). If \( x \) is the negative number of least magnitude in the type of \( x \), nextup(\( x \)) is \( -0 \) if the type has signed zeros and is \( 0 \) otherwise. If \( x \) is zero, nextup(\( x \)) is the positive number of least magnitude in the type of \( x \). nextup(HUGE_VAL) is HUGE_VAL.

Returns

3 The nextup functions return the next representable value in the specified type greater than \( x \).

7.12.11.6 The nextdown functions

Synopsis

```c
#include <math.h>

double nextdown(double x);
float nextdownf(float x);
```

\(^{255}\)The argument values are converted to the type of the function, even by a macro implementation of the function.

\(^{256}\)The result of the nexttoward functions is determined in the type of the function, without loss of range or precision in a floating second argument.
long double nextdown(long double x);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal128 nextdownd128(_Decimal128 x);
 _Decimal64 nextdownd64(_Decimal64 x);
 _Decimal32 nextdownd32(_Decimal32 x);
#endif

Description

2 The `nextdown` functions determine the next representable value, in the type of the function, less than x. If x is the positive number of least magnitude in the type of x, `nextdown(x)` is +0 if the type has signed zeros and is 0 otherwise. If x is zero, `nextdown(x)` is the negative number of least magnitude in the type of x. `nextdown(-HUGE_VAL)` is -HUGE_VAL.

Returns

3 The `nextdown` functions return the next representable value in the specified type less than x.

7.12.11.7 The canonicalize functions

Synopsis

```c
#include <math.h>
int canonicalize(double * cx, const double * x);
int canonicalizef(float * cx, const float * x);
int canonicalizel(long double * cx, const long double * x);
#ifdef __STDC_IEC_60559_DFP__
int canonicalized32(_Decimal32 cx, const _Decimal32 * x);
int canonicalized64(_Decimal64 cx, const _Decimal64 * x);
int canonicalized128(_Decimal128 cx, const _Decimal128 * x);
#endif
```

Description

2 The `canonicalize` functions attempt to produce a canonical version of the floating-point representation in the object pointed to by the argument x, as if to a temporary object of the specified type, and store the canonical result in the object pointed to by the argument `cx`. If the input x is a signaling NaN, the `canonicalize` functions are intended to store a canonical quiet NaN. If a canonical result is not produced the object pointed to by `cx` is unchanged.

Returns

3 The `canonicalize` functions return zero if a canonical result is stored in the object pointed to by `cx`. Otherwise they return a nonzero value.

7.12.12 Maximum, minimum, and positive difference functions

7.12.12.1 The `fdim` functions

Synopsis

```c
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);
 _Decimal64 fdimd64(_Decimal64 x, _Decimal64 y);
 _Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);
#endif
```

The arguments x and cx may point to the same object.
Description

The `fdim` functions determine the positive difference between their arguments:

\[
\begin{align*}
& x - y & \text{if } x > y \\
& +0 & \text{if } x \leq y
\end{align*}
\]

A range error may occur.

Returns

The `fdim` functions return the positive difference value.

7.12.12.2 The `fmax` functions

Synopsis replace Synopsis

```c
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal32 fmaxd32(_Decimal32 x, _Decimal32 y);
 _Decimal64 fmaxd64(_Decimal64 x, _Decimal64 y);
 _Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

The `fmax` functions determine the maximum numeric value of their arguments.

Returns

The `fmax` functions return the maximum numeric value of their arguments.

7.12.12.3 The `fmin` functions

Synopsis replace Synopsis

```c
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
 _Decimal32 fmind32(_Decimal32 x, _Decimal32 y);
 _Decimal64 fmind64(_Decimal64 x, _Decimal64 y);
 _Decimal128 fmind128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

The `fmin` functions determine the minimum numeric value of their arguments.

Returns

The `fmin` functions return the minimum numeric value of their arguments.

7.12.12.4 The `fmaxmag` functions

Synopsis

```c
#include <math.h>

double fmaxmag(double x, double y);
float fmaxmagf(float x, float y);
long double fmaxmagl(long double x, long double y);
```

258) Quiet NaN arguments are treated as missing data: if one argument is a quiet NaN and the other numeric, then the `fmax` functions choose the numeric value. See F.10.9.2.

259) The `fmin` functions are analogous to the `fmax` functions in their treatment of quiet NaNs.
The `fmaxmag` functions determine the value of their argument whose magnitude is the maximum of the magnitudes of the arguments: the value of \( x \) if \( |x| > |y| \), \( y \) if \( |x| < |y| \), and `fmax(x, y)` otherwise.

**Returns**

The `fmaxmag` functions return the value of their argument of maximum magnitude.

### Synopsis 7.12.12.5 The `fminmag` functions

```c
#include <math.h>
double fminmag(double x, double y);
float fminmag(float x, float y);
long double fminmag(long double x, long double y);
#endif
_Decimal32 fminmag32(_Decimal32 x, _Decimal32 y);
_Decimal64 fminmag64(_Decimal64 x, _Decimal64 y);
_Decimal128 fminmag128(_Decimal128 x, _Decimal128 y);
#endif
```

**Description**

The `fminmag` functions determine the value of their argument whose magnitude is the minimum of the magnitudes of the arguments: the value of \( x \) if \( |x| < |y| \), \( y \) if \( |x| > |y| \), and `fmin(x, y)` otherwise.

**Returns**

The `fminmag` functions return the value of their argument of minimum magnitude.

### Floating multiply-add

#### Synopsis 7.12.13.1 The `fma` functions

```c
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
#endif
_Decimal32 fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
_Decimal64 fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);
#endif
```

**Description**

The `fma` functions compute \((x \times y) + z\), rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur.

**Returns**

The `fma` functions return \((x \times y) + z\), rounded as one ternary operation.

---

260] Quiet NaN arguments are treated as missing data: if one argument is a quiet NaN and the other numeric, then the `fmaxmag` functions choose the numeric value. See 10.9.4.  
261] The `fminmag` functions are analogous to the `fmaxmag` functions in their treatment of quiet NaNs.
7.12.14 Functions that round result to narrower type

The functions in this subclause round their results to a type typically narrower\(^{262}\) than the parameter types.

7.12.14.1 Add and round to narrower type

**Synopsis**

```c
#include <math.h>
float fadd(double x, double y);
float faddl(long double x, long double y);
double daddl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
  __Decimal32 d32addd64(__Decimal64 x, __Decimal64 y);
  __Decimal32 d32addd128(__Decimal128 x, __Decimal128 y);
  __Decimal64 d64addd128(__Decimal128 x, __Decimal128 y);
#endif
```

**Description**

These functions compute the sum of \(x + y\), rounded to the type of the function. They compute the sum (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for infinite arguments.

**Returns**

These functions return the sum of \(x + y\), rounded to the type of the function.

7.12.14.2 Subtract and round to narrower type

**Synopsis**

```c
#include <math.h>
float fsub(double x, double y);
float fsubl(long double x, long double y);
double dsubl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
  __Decimal32 d32subd64(__Decimal64 x, __Decimal64 y);
  __Decimal32 d32subd128(__Decimal128 x, __Decimal128 y);
  __Decimal64 d64subd128(__Decimal128 x, __Decimal128 y);
#endif
```

**Description**

These functions compute the difference of \(x - y\), rounded to the type of the function. They compute the difference (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for infinite arguments.

**Returns**

These functions return the difference of \(x - y\), rounded to the type of the function.

7.12.14.3 Multiply and round to narrower type

**Synopsis**

```c
#include <math.h>
float fmul(double x, double y);
float fmull(long double x, long double y);
double dmull(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
  __Decimal32 d32muld64(__Decimal64 x, __Decimal64 y);
#endif
```

\(^{262}\)In some cases the destination type might not be narrower than the parameter types. For example, double might not be narrower than long double.
Description

These functions compute the product \( x \times y \), rounded to the type of the function. They compute the product (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error occurs for one infinite argument and one zero argument.

Returns

These functions return the product of \( x \times y \), rounded to the type of the function.

7.12.14.4 Divide and round to narrower type

Synopsis

```c
#include <math.h>
float fdiv(double x, double y);
float fdivl(long double x, long double y);
double ddivl(long double x, long double y);
#define __STDC_IEC_60559_DFP__
_DECIMAL32 d32divd128(_DECIMAL128 x, _DECIMAL128 y);
_DECIMAL32 d32divd64(_DECIMAL128 x, _DECIMAL128 y);
#undef __STDC_IEC_60559_DFP__
```

Description

These functions compute the quotient \( x \div y \), rounded to the type of the function. They compute the quotient (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error occurs for either both arguments infinite or both arguments zero. A pole error occurs for a finite \( x \) and a zero \( y \).

Returns

These functions return the quotient \( x \div y \), rounded to the type of the function.

7.12.14.5 Floating point multiply-add and round to narrower type

Synopsis

```c
#include <math.h>
float ffma(double x, double y, double z);
float ffmal(long double x, long double y, long double z);
double dfmal(long double x, long double y, long double z);
#define __STDC_IEC_60559_DFP__
_DECIMAL32 d32fmad64(_DECIMAL64 x, _DECIMAL64 y, _DECIMAL64 z);
_DECIMAL32 d32fmad128(_DECIMAL128 x, _DECIMAL128 y, _DECIMAL128 z);
#undef __STDC_IEC_60559_DFP__
```

Description

These functions compute \((x \times y) + z\), rounded to the type of the function. They compute \((x \times y) + z\) (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for an infinite argument.

Returns

These functions return \((x \times y) + z\), rounded to the type of the function.

7.12.14.6 Square root rounded to narrower type

Synopsis
**Synopsis**

```c
#include <math.h>
#endif

_Bool samequantum32(_Decimal32 x, _Decimal32 y);
_Bool samequantum64(_Decimal64 x, _Decimal64 y);
_Bool samequantum128(_Decimal128 x, _Decimal128 y);
#endif
```

**Description**

3 The *samequantumN* functions return a value with the numerical value of *x* (except for any rounding) and the quantum exponent of *y*.

7.12.15.2 The *samequantumN* functions

**Synopsis**

```c
#include <math.h>
#endif

_Bool samequantum32(_Decimal32 x, _Decimal32 y);
_Bool samequantum64(_Decimal64 x, _Decimal64 y);
_Bool samequantum128(_Decimal128 x, _Decimal128 y);
#endif
```

**Description**

3 The *samequantumN* functions return a value with the numerical value of *x* (except for any rounding) and the quantum exponent of *y*.
The `samequantumquantum` functions determine if the quantum exponents of \( x \) and \( y \) are the same. If both \( x \) and \( y \) are NaN, or both infinite, they have the same quantum exponents; if exactly one operand is infinite or exactly one operand is NaN, they do not have the same quantum exponents. The `samequantumquantum` functions raise no floating-point exception.

Returns

The `samequantumquantum` functions return nonzero (true) when \( x \) and \( y \) have the same quantum exponents, zero (false) otherwise.

### 7.12.15.3 The `quantumquantum` functions

**Synopsis**

```c
#include <math.h>
#endif
__STDC_IEC_60559_DFP__

_Decimal32 quantumd32(_Decimal32 x);  
_Decimal64 quantumd64(_Decimal64 x);  
_Decimal128 quantumd128(_Decimal128 x);  
#endif
```

**Description**

The `quantumquantum` functions compute the quantum (5.2.4.2.3) of a finite argument. If \( x \) is infinite, the result is +\( \infty \).

Returns

The `quantumquantum` functions return the quantum of \( x \).

### 7.12.15.4 The `llquantexpdquantum` functions

**Synopsis**

```c
#include <math.h>
#endif
__STDC_IEC_60559_DFP__

long long int llquantexpd32(_Decimal32 x);  
long long int llquantexpd64(_Decimal64 x);  
long long int llquantexpd128(_Decimal128 x);  
#endif
```

**Description**

The `llquantexpdquantum` functions compute the quantum exponent (5.2.4.2.3) of a finite argument. If \( x \) is infinite or NaN, they compute \texttt{LLONG_MIN}, the “invalid” floating-point exception is raised, and a domain error occurs.

Returns

The `llquantexpdquantum` functions return the quantum exponent of \( x \).

### 7.12.16 Decimal re-encoding functions

IEC 60559 specifies two different schemes to encode significands in the object representation of a decimal floating-point object: one based on decimal encoding (which packs three decimal digits into 10 bits), the other based on binary encoding (as a binary integer). An implementation may use either of these encoding schemes for its decimal floating types. The re-encoding functions in this subclause provide conversions between external decimal data with a given encoding scheme and the implementation’s corresponding decimal floating type.

#### 7.12.16.1 The `encodedecdquantum` functions

**Synopsis**

```c
#include <math.h>
#endif
__STDC_IEC_60559_DFP__

void encodedecd32(unsigned char encptr[restrict static 4],  
                 const _Decimal32 *restrict xptr);  
void encodedecd64(unsigned char encptr[restrict static 8],
                 const _Decimal64 *restrict xptr);
```
const _Decimal64* restrict xptr);
void encodedecd128(unsigned char* encptr[restrict static 16],
const _Decimal128* restrict xptr);
#endif

Description

The encodedecdN functions convert *xptr into an IEC 60559 decimalN encoding in the encoding scheme based on decimal encoding of the significand and store the resulting encoding as an N/8 element array, with 8 bits per array element, in the object pointed to by encptr. The order of bytes in the array is implementation-defined. These functions preserve the value of *xptr and raise no floating-point exceptions. If *xptr is non-canonical, these functions may or may not produce a canonical encoding.

Returns

The encodedecdN functions return no value.

7.12.16.2 The decodedecdN functions

Synopsis

#include <math.h>
#ifndef __STDC_IEC_60559_DFP__
void decodedecd32(_Decimal32* restrict xptr,
const unsigned char* encptr[restrict static 4]);
void decodedecd64(_Decimal64* restrict xptr,
const unsigned char* encptr[restrict static 8]);
void decodedecd128(_Decimal128* restrict xptr,
const unsigned char* encptr[restrict static 16]);
#endif

Description

The decodedecdN functions interpret the N/8 element array pointed to by encptr as an IEC 60559 decimalN encoding, with 8 bits per array element, in the encoding scheme based on decimal encoding of the significand. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a value of the decimal floating type, and store the result in the object pointed to by xptr. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns

The decodedecdN functions return no value.

7.12.16.3 The encodebindN functions

Synopsis

#include <math.h>
#ifndef __STDC_IEC_60559_DFP__
void encodebind32(unsigned char* encptr[restrict static 4],
const _Decimal32* restrict xptr);
void encodebind64(unsigned char* encptr[restrict static 8],
const _Decimal64* restrict xptr);
void encodebind128(unsigned char* encptr[restrict static 16],
const _Decimal128* restrict xptr);
#endif

Description

The encodebindN functions convert *xptr into an IEC 60559 decimalN encoding in the encoding scheme based on binary encoding of the significand and store the resulting encoding as an N/8 element array, with 8 bits per array element, in the object pointed to by encptr. The order of bytes
in the array is implementation-defined. These functions preserve the value of \*xptr and raise no floating-point exceptions. If \*xptr is non-canonical, these functions may or may not produce a canonical encoding.

Returns
3 The `decodebindN` functions return no value.

### 7.12.16.4 The `decodebindN` functions

#### Synopsis

```c
#include <math.h>

if (STDC_IEC_60559_DFP)
    void decodebind32(_Decimal32 * restrict xptr,
                       const unsigned char encptr[restrict static 4]);

if (STDC_IEC_60559_DFP)
    void decodebind64(_Decimal64 * restrict xptr,
                       const unsigned char encptr[restrict static 8]);

if (STDC_IEC_60559_DFP)
    void decodebind128(_Decimal128 * restrict xptr,
                       const unsigned char encptr[restrict static 16]);
#endif
```

#### Description

2 The `decodebindN` functions interpret the \(N/8\) element array pointed to by `encptr` as an IEC 60559 decimal\(N\) encoding, with 8 bits per array element, in the encoding scheme based on binary encoding of the significand. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a value of decimal floating type, and store the result in the object pointed to by `xptr`. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns
3 The `decodebindN` functions return no value.

### 7.12.17 Comparison macros

1 The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships — `less`, `greater`, and `equal` — is true. Relational operators may raise the “invalid” floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the `unordered` relationship is true.\(^{263}\) The following subclauses Subclauses 7.12.17.1 through 7.12.17.6 provide macros that are `quiet` (non-floating-point exception raising) `quiet` versions of the relational operators, and other comparison macros that: the macros do not raise the “invalid” floating-point exception as an effect of quiet NaN arguments. The comparison macros facilitate writing efficient code that accounts for quiet NaN NaNs \(-\) without suffering the “invalid” floating-point exception. In the synopses in this subclause, `real-floating` indicates that the argument shall be an expression of real floating type\(^ {264}\) (both arguments need not have the same type).\(^ {265}\) If either argument has decimal floating type, the other argument shall have decimal floating type as well.

#### 7.12.17.1 The `isgreater` macro

##### Synopsis

```c
#include <math.h>

int isgreater(real-floating x, real-floating y);
```

---

\(^{263}\)IEC 60559 requires that the built-in relational operators raise the “invalid” floating-point exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

\(^{264}\)If any argument is of integer type, or any other type that is not a real floating type, the behavior is undefined.

\(^{265}\)Whether an argument represented in a format wider than its semantic type is converted to the semantic type is unspecified.
Description
2 The `isgreater` macro determines whether its first argument is greater than its second argument. The value of `isgreater(x, y)` is always equal to `(x) > (y)`; however, unlike `(x) > (y)`, `isgreater(x, y)` does not raise the “invalid” floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns
3 The `isgreater` macro returns the value of `(x) > (y)`.

7.12.17.2 The `isgreaterequal` macro

Synopsis
1
```c
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

Description
2 The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x, y)` is always equal to `(x) >= (y)`; however, unlike `(x) >= (y)`, `isgreaterequal(x, y)` does not raise the “invalid” floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns
3 The `isgreaterequal` macro returns the value of `(x) >= (y)`.

7.12.17.3 The `isless` macro

Synopsis
1
```c
#include <math.h>
int isless(real-floating x, real-floating y);
```

Description
2 The `isless` macro determines whether its first argument is less than its second argument. The value of `isless(x, y)` is always equal to `(x) < (y)`; however, unlike `(x) < (y)`, `isless(x, y)` does not raise the “invalid” floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns
3 The `isless` macro returns the value of `(x) < (y)`.

7.12.17.4 The `islessequal` macro

Synopsis
1
```c
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

Description
2 The `islessequal` macro determines whether its first argument is less than or equal to its second argument. The value of `islessequal(x, y)` is always equal to `(x) <= (y)`; however, unlike `(x) <= (y)`, `islessequal(x, y)` does not raise the “invalid” floating-point exception when `x` and `y` are unordered and neither is a signaling NaN.

Returns
3 The `islessequal` macro returns the value of `(x) <= (y)`.

7.12.17.5 The `islessgreater` macro

Synopsis
1
```c
#include <math.h>
int islessgreater(real-floating x, real-floating y);
```
Description
2 The \texttt{islessgreater} macro determines whether its first argument is less than or greater than its second argument. The \texttt{islessgreater}(x, y) macro is similar to \((x) < (y) || (x) > (y)\); however, \texttt{islessgreater}(x, y) does not raise the “invalid” floating-point exception when \(x\) and \(y\) are unordered and neither is a signaling NaN (nor does it evaluate \(x\) and \(y\) twice).

Returns
3 The \texttt{islessgreater} macro returns the value of \((x) < (y) || (x) > (y)\).

7.12.17.6 The \texttt{isunordered} macro

Synopsis
1
\begin{verbatim}
#include <math.h>

int isunordered(real-floating x, real-floating y);
\end{verbatim}

Description
2 The \texttt{isunordered} macro determines whether its arguments are unordered. It raises no floating-point exceptions if neither argument is a signaling NaN.

Returns
3 The \texttt{isunordered} macro returns 1 if its arguments are unordered and 0 otherwise.

7.12.17.7 The \texttt{iseqsig} macro

Synopsis
1
\begin{verbatim}
#include <math.h>

int iseqsig(real-floating x, real-floating y);
\end{verbatim}

Description
2 The \texttt{iseqsig} macro determines whether its arguments are equal. If an argument is a NaN, a domain error occurs for the macro, as if a domain error occurred for a function (7.12.1).

Returns
3 The \texttt{iseqsig} macro returns 1 if its arguments are equal and 0 otherwise.
7.13 Nonlocal jumps <setjmp.h>

The header <setjmp.h> defines the macro `setjmp`, and declares one function and one type, for bypassing the normal function call and return discipline.\(^{266}\)

The type declared is

```
jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to the `setjmp` macro consists of information sufficient for a call to the `longjmp` function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name `setjmp`, the behavior is undefined.

7.13.1 Save calling environment

7.13.1.1 The `setjmp` macro

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later use by the `longjmp` function.

Returns

If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` macro returns a nonzero value.

Environmental limits

An invocation of the `setjmp` macro shall appear only in one of the following contexts:

- the entire controlling expression of a selection or iteration statement;
- one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;
- the operand of a unary `!` operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or
- the entire expression of an expression statement (possibly cast to `void`).

If the invocation appears in any other context, the behavior is undefined.

7.13.2 Restore calling environment

7.13.2.1 The `longjmp` function

Synopsis

```
#include <setjmp.h>
_Noreturn void longjmp(jmp_buf env, int val);
```

\(^{266}\)These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.
Description
2 The longjmp function restores the environment saved by the most recent invocation of the setjmp macro in the same invocation of the program with the corresponding jmp_buf argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the setjmp macro has terminated execution\(^{267}\) in the interim, or if the invocation of the setjmp macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

3 All accessible objects have values, and all other components of the abstract machine\(^ {268}\) have state, as of the time the longjmp function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the setjmp invocation and longjmp call are indeterminate.

Returns
4 After longjmp is completed, thread execution continues as if the corresponding invocation of the setjmp macro had just returned the value specified by val. The longjmp function cannot cause the setjmp macro to return the value 0; if val is 0, the setjmp macro returns the value 1.

EXAMPLE The longjmp function that returns control back to the point of the setjmp invocation might cause memory associated with a variable length array object to be squandered.

```c
#include <setjmp.h>
jmp_buf buf;

void g(int n);
void h(int n);
int n = 6;

void f(void)
{
    int x[n]; // valid: f is not terminated
    setjmp(buf);
    g(n);
}

void g(int n)
{
    int a[n]; // a may remain allocated
    h(n);
}

void h(int n)
{
    int b[n]; // b may remain allocated
    longjmp(buf, 2); // might cause memory loss
}
```

\(^{267}\)For example, by executing a return statement or because another longjmp call has caused a transfer to a setjmp invocation in a function earlier in the set of nested calls.

\(^{268}\)This includes, but is not limited to, the floating-point status flags and the state of open files.
7.14 Signal handling `<signal.h>`

The header `<signal.h>` declares a type and two functions and defines several macros, for handling various signals (conditions that may be reported during program execution).

The type defined is

```
sig_atomic_t
```

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The macros defined are

```
SIG_DFL
SIG_ERR
SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the `signal` function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type `int` and distinct values that are the signal numbers, each corresponding to the specified condition:

- **SIGABRT**: abnormal termination, such as is initiated by the `abort` function
- **SIGFPE**: an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow
- **SIGILL**: detection of an invalid function image, such as an invalid instruction
- **SIGINT**: receipt of an interactive attention signal
- **SIGSEGV**: an invalid access to storage
- **SIGTERM**: a termination request sent to the program

An implementation need not generate any of these signals, except as a result of explicit calls to the `raise` function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters `SIG` and an uppercase letter or with `SIG_` and an uppercase letter,\(^{269}\) may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

7.14.1 Specify signal handling

7.14.1.1 The `signal` function

**Synopsis**

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

**Description**

The `signal` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise, `func` shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library),\(^{270}\) is called a **signal handler**.

\(^{269}\)See “future library directions” (7.31.9). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

\(^{270}\)This includes functions called indirectly via standard library functions (e.g., a `SIGABRT` handler called via the `abort` function).
When a signal occurs and *func* points to a function, it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of `SIGILL`, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of `sig` is `SIGFPE`, `SIGILL`, `SIGSEGV`, or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

If the signal occurs as the result of calling the `abort` or `raise` function, the signal handler shall not call the `raise` function.

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or the signal handler calls any function in the standard library other than

- the `abort` function,
- the `_Exit` function,
- the `quick_exit` function,
- the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,
- the `atomic_is_lock_free` function with any atomic argument, or
- the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the `signal` function results in a `SIG_ERR` return, the value of `errno` is indeterminate.

At program startup, the equivalent of `signal(sig, SIG_IGN);` may be executed for some signals selected in an implementation-defined manner; the equivalent of `signal(sig, SIG_DFL);` is executed for all other signals defined by the implementation.

Use of this function in a multi-threaded program results in undefined behavior. The implementation shall behave as if no library function calls the `signal` function.

**Returns**

If the request can be honored, the `signal` function returns the value of `func` for the most recent successful call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

**Forward references:** the `abort` function (7.22.4.1), the `exit` function (7.22.4.4), the `_Exit` function (7.22.4.5), the `quick_exit` function (7.22.4.7).

### 7.14.2 Send signal

#### 7.14.2.1 The `raise` function

**Synopsis**

```c
#include <signal.h>
int raise(int sig);
```

If any signal is generated by an asynchronous signal handler, the behavior is undefined.
Description
2 The \texttt{raise} function carries out the actions described in 7.14.1.1 for the signal \texttt{sig}. If a signal handler is called, the \texttt{raise} function shall not return until after the signal handler does.

Returns
3 The \texttt{raise} function returns zero if successful, nonzero if unsuccessful.
7.15 Alignment <stdalign.h>

1 The header <stdalign.h> defines four macros.

2 The macro

   alignas

expands to _Alignas; the macro

   alignof

expands to _Alignof.

3 The remaining macros are suitable for use in #if preprocessing directives. They are

   __alignas_is_defined

and

   __alignof_is_defined

which both expand to the integer constant 1.
7.16 Variable arguments <stdarg.h>

The header <stdarg.h> declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function may be called with a variable number of arguments of varying types. As described in 6.9.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated parmN in this description.

The type declared is

```
va_list
```

which is a complete object type suitable for holding information needed by the macros `va_start`, `va_arg`, `va_end`, and `va_copy`. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type `va_list`. The object ap may be passed as an argument to another function; if that function invokes the `va_arg` macro with parameter ap, the value of ap in the calling function is indeterminate and shall be passed to the `va_end` macro prior to any further reference to ap.

7.16.1 Variable argument list access macros

The `va_start` and `va_arg` macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether `va_copy` and `va_end` are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the `va_start` and `va_copy` macros shall be matched by a corresponding invocation of the `va_end` macro in the same function.

7.16.1.1 The `va_arg` macro

**Synopsis**

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

**Description**

The `va_arg` macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter ap shall have been initialized by the `va_start` or `va_copy` macro (without an intervening invocation of the `va_end` macro for the same ap). Each invocation of the `va_arg` macro modifies ap so that the values of successive arguments are returned in turn. The parameter type shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to type. If there is no actual next argument, or if type is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to `void` and the other is a pointer to a character type.

**Returns**

The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the argument after that specified by parmN. Successive invocations return the values of the remaining arguments in succession.

---

272 It is permitted to create a pointer to a `va_list` and pass that pointer to another function, in which case the original function can make further use of the original list after the other function returns.
7.16.1.2 The va_copy macro

Synopsis

```c
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

The va_copy macro initializes dest as a copy of src, as if the va_start macro had been applied to dest followed by the same sequence of uses of the va_arg macro as had previously been used to reach the present state of src. Neither the va_copy nor va_start macro shall be invoked to reinitialize dest without an intervening invocation of the va_end macro for the same dest.

Returns

The va_copy macro returns no value.

7.16.1.3 The va_end macro

Synopsis

```c
#include <stdarg.h>
void va_end(va_list ap);
```

Description

The va_end macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the va_start macro, or the function containing the expansion of the va_copy macro, that initialized the va_list ap. The va_end macro may modify ap so that it is no longer usable (without being reinitialized by the va_start or va_copy macro). If there is no corresponding invocation of the va_start or va_copy macro, or if the va_end macro is not invoked before the return, the behavior is undefined.

Returns

The va_end macro returns no value.

7.16.1.4 The va_start macro

Synopsis

```c
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

The va_start macro shall be invoked before any access to the unnamed arguments.

The va_start macro initializes ap for subsequent use by the va_arg and va_end macros. Neither the va_start nor va_copy macro shall be invoked to reinitialize ap without an intervening invocation of the va_end macro for the same ap.

The parameter parmN is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ... ). If the parameter parmN is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

The va_start macro returns no value.

EXAMPLE 1 The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.
Each call to `f1` is required to have visible the definition of the function or a declaration such as

```c
void f1(int n_ptrs, ...);
```

**EXAMPLE 2** The function `f3` is similar, but saves the status of the variable argument list after the indicated number of arguments; after `f2` has been called once with the whole list, the trailing part of the list is gathered again and passed to function `f4`.

```c
#include <stdarg.h>
define MAXARGS 31

void f3(int n_ptrs, int f4_after, ...) {
    va_list ap, ap_save;
    char *array[MAXARGS];
    int ptr_no = 0;

    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, f4_after);
    while (ptr_no < n_ptrs) {
        array[ptr_no++] = va_arg(ap, char *);
    }
    va_end(ap);

    f2(n_ptrs, array);

    // Now process the saved copy.
    n_ptrs -= f4_after;
    ptr_no = 0;
    while (ptr_no < n_ptrs)
        array[ptr_no++] = va_arg(ap_save, char *);
    va_end(ap_save);
    f4(n_ptrs, array);
}
```
7.17 Atomics `<stdatomic.h>`

7.17.1 Introduction

The header `<stdatomic.h>` defines several macros and declares several types and functions for performing atomic operations on data shared between threads.\(^{273}\)

Implementations that define the macro `__STDC_NO_ATOMICS__` need not provide this header nor support any of its facilities.

The macros defined are the atomic lock-free macros

```
ATOMIC_BOOL_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
```

which expand to constant expressions suitable for use in `#if` preprocessing directives and which indicate the lock-free property of the corresponding atomic types (both signed and unsigned); and

```
ATOMIC_FLAG_INIT
```

which expands to an initializer for an object of type `atomic_flag`.

The types include

```
memory_order
```

which is an enumerated type whose enumerators identify memory ordering constraints;

```
atomic_flag
```

which is a structure type representing a lock-free, primitive atomic flag; and several atomic analogs of integer types.

In the following synopses:

— An `A` refers to an atomic type.

— A `C` refers to its corresponding non-atomic type.

— An `M` refers to the type of the other argument for arithmetic operations. For atomic integer types, `M` is `C`. For atomic pointer types, `M` is `ptrdiff_t`.

— The functions not ending in `_explicit` have the same semantics as the corresponding `_explicit` function with `memory_order_seq_cst` for the `memory_order` argument.

It is unspecified whether any generic function declared in `<stdatomic.h>` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is undefined.

**NOTE** Many operations are volatile-qualified. The “volatile as device register” semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects.

\(^{273}\)See “future library directions” (7.31.10).
7.17.2 Initialization

7.17.2.1 The ATOMIC_VAR_INIT macro

Synopsis

```c
#include <stdatomic.h>
#define ATOMIC_VAR_INIT(C value)
```

Description

The ATOMIC_VAR_INIT macro expands to a token sequence suitable for initializing an atomic object of a type that is initialization-compatible with `value`. An atomic object with automatic storage duration that is not explicitly initialized is initially in an indeterminate state; however, the default (zero) initialization for objects with static or thread-local storage duration is guaranteed to produce a valid state.\(^{274}\)

Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

EXAMPLE

```c
atomic_int guide = ATOMIC_VAR_INIT(42);
```

7.17.2.2 The atomic_init generic function

Synopsis

```c
#include <stdatomic.h>
void atomic_init(void volatile *obj, C value);
```

Description

The atomic_init generic function initializes the atomic object pointed to by `obj` to the value `value`, while also initializing any additional state that the implementation might need to carry for the atomic object.

Although this function initializes an atomic object, it does not avoid data races; concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race.

If a signal occurs other than as the result of calling the abort or raise functions, the behavior is undefined if the signal handler calls the atomic_init generic function.

Returns

The atomic_init generic function returns no value.

EXAMPLE

```c
atomic_int guide;
atomic_init(&guide, 42);
```

7.17.3 Order and consistency

The enumerated type memory_order specifies the detailed regular (non-atomic) memory synchronization operations as defined in 5.1.2.4 and may provide for operation ordering. Its enumeration constants are as follows:\(^{275}\)

```c
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acquire
memory_order_seq_cst
```

\(^{274}\)See “future library directions” (7.31.10).

\(^{275}\)See “future library directions” (7.31.10).
For **memory_order_relaxed**, no operation orders memory.

For **memory_order_release**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a store operation performs a release operation on the affected memory location.

For **memory_order_acquire**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a load operation performs an acquire operation on the affected memory location.

For **memory_order_consume**, a load operation performs a consume operation on the affected memory location.

There shall be a single total order \( S \) on all **memory_order_seq_cst** operations, consistent with the “happens before” order and modification orders for all affected locations, such that each **memory_order_seq_cst** operation \( B \) that loads a value from an atomic object \( M \) observes one of the following values:

- the result of the last modification \( A \) of \( M \) that precedes \( B \) in \( S \), if it exists, or
- if \( A \) exists, the result of some modification of \( M \) that is not **memory_order_seq_cst** and that does not happen before \( A \), or
- if \( A \) does not exist, the result of some modification of \( M \) that is not **memory_order_seq_cst**.

**NOTE 1** Although it is not explicitly required that \( S \) include lock operations, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the “happens before” ordering.

**NOTE 2** Atomic operations specifying **memory_order_relaxed** are relaxed only with respect to memory ordering. Implementations still guarantee that any given atomic access to a particular atomic object is indivisible with respect to all other atomic accesses to that object.

For an atomic operation \( B \) that reads the value of an atomic object \( M \), if there is a **memory_order_seq_cst** fence \( X \) sequenced before \( B \), then \( B \) observes either the last **memory_order_seq_cst** modification of \( M \) preceding \( X \) in the total order \( S \) or a later modification of \( M \) in its modification order.

For atomic operations \( A \) and \( B \) on an atomic object \( M \), where \( A \) modifies \( M \) and \( B \) takes its value, if there is a **memory_order_seq_cst** fence \( X \) such that \( A \) is sequenced before \( X \) and \( B \) follows \( X \) in \( S \), then \( B \) observes either the effects of \( A \) or a later modification of \( M \) in its modification order.

For atomic modifications \( A \) and \( B \) of an atomic object \( M \), \( B \) occurs later than \( A \) in the modification order of \( M \) if:

- there is a **memory_order_seq_cst** fence \( X \) such that \( A \) is sequenced before \( X \), and \( X \) precedes \( B \) in \( S \), or
- there is a **memory_order_seq_cst** fence \( Y \) such that \( Y \) is sequenced before \( B \), and \( A \) precedes \( Y \) in \( S \), or
- there are **memory_order_seq_cst** fences \( X \) and \( Y \) such that \( A \) is sequenced before \( X \), \( Y \) is sequenced before \( B \), and \( X \) precedes \( Y \) in \( S \).

**§ 7.17.3 Library**

Atomic read-modify-write operations shall always read the last value (in the modification order) stored before the write associated with the read-modify-write operation.

An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence. The ordering of evaluations in this sequence shall be such that

- If an evaluation \( B \) observes a value computed by \( A \) in a different thread, then \( B \) does not happen before \( A \).
- If an evaluation \( A \) is included in the sequence, then all evaluations that assign to the same variable and happen before \( A \) are also included.
NOTE 3 The second requirement disallows “out-of-thin-air”, or “speculative” stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations can appear in this sequence out of thread order. For example, with \( x \) and \( y \) initially zero,

```c
// Thread 1:
    r1 = atomic_load_explicit(&y, memory_order_relaxed);
    atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
    r2 = atomic_load_explicit(&x, memory_order_relaxed);
    atomic_store_explicit(&y, 42, memory_order_relaxed);
```

is allowed to produce \( r1 == 42 \&\& r2 == 42 \). The sequence of evaluations justifying this consists of:

```c
atomic_store_explicit(&y, 42, memory_order_relaxed);
    r1 = atomic_load_explicit(&y, memory_order_relaxed);
    atomic_store_explicit(&x, r1, memory_order_relaxed);
    r2 = atomic_load_explicit(&x, memory_order_relaxed);
```

On the other hand,

```c
// Thread 1:
    r1 = atomic_load_explicit(&y, memory_order_relaxed);
    atomic_store_explicit(&x, r1, memory_order_relaxed);

// Thread 2:
    r2 = atomic_load_explicit(&x, memory_order_relaxed);
    atomic_store_explicit(&y, r2, memory_order_relaxed);
```

is not allowed to produce \( r1 == 42 \&\& r2 == 42 \), since there is no sequence of evaluations that results in the computation of 42. In the absence of “relaxed” operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact.

**Recommended practice**

The requirements do not forbid \( r1 == 42 \&\& r2 == 42 \) in the following example, with \( x \) and \( y \) initially zero:

```c
// Thread 1:
    r1 = atomic_load_explicit(&x, memory_order_relaxed);
    if (r1 == 42)
        atomic_store_explicit(&y, r1, memory_order_relaxed);

// Thread 2:
    r2 = atomic_load_explicit(&y, memory_order_relaxed);
    if (r2 == 42)
        atomic_store_explicit(&x, 42, memory_order_relaxed);
```

However, this is not useful behavior, and implementations should not allow it.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

**7.17.3.1 The kill_dependency macro**

**Synopsis**

```c
#include <stdatomic.h>

type kill_dependency(type y);
```

**Description**

The `kill_dependency` macro terminates a dependency chain; the argument does not carry a dependency to the return value.
Returns
3 The \texttt{kill\_dependency} macro returns the value of \( y \).

7.17.4 Fences
1 This subclause introduces synchronization primitives called \textit{fences}. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an \textit{acquire fence}; a fence with release semantics is called a \textit{release fence}.

2 A release fence \( A \) synchronizes with an acquire fence \( B \) if there exist atomic operations \( X \) and \( Y \), both operating on some atomic object \( M \), such that \( A \) is sequenced before \( X \), \( X \) modifies \( M \), \( Y \) is sequenced before \( B \), and \( Y \) reads the value written by \( X \) or a value written by any side effect in the hypothetical release sequence \( X \) would head if it were a release operation.

3 A release fence \( A \) synchronizes with an atomic operation \( B \) that performs an acquire operation on an atomic object \( M \) if there exists an atomic operation \( X \) such that \( A \) is sequenced before \( X \), \( X \) modifies \( M \), and \( B \) reads the value written by \( X \) or a value written by any side effect in the hypothetical release sequence \( X \) would head if it were a release operation.

4 An atomic operation \( A \) that is a release operation on an atomic object \( M \) synchronizes with an acquire fence \( B \) if there exists some atomic operation \( X \) on \( M \) such that \( X \) is sequenced before \( B \) and reads the value written by \( A \) or a value written by any side effect in the release sequence headed by \( A \).

7.17.4.1 The \texttt{atomic\_thread\_fence} function

Synopsis
1

\begin{verbatim}
#include <stdatomic.h>
void atomic_thread_fence(memory_order order);
\end{verbatim}

Description
2 Depending on the value of \texttt{order}, this operation:
   
   — has no effects, if \texttt{order == memory\_order\_relaxed};
   
   — is an acquire fence, if \texttt{order == memory\_order\_acquire} or \texttt{order == memory\_order\_consume};
   
   — is a release fence, if \texttt{order == memory\_order\_release};
   
   — is both an acquire fence and a release fence, if \texttt{order == memory\_order\_acq\_rel};
   
   — is a sequentially consistent acquire and release fence, if \texttt{order == memory\_order\_seq\_cst}.

Returns
3 The \texttt{atomic\_thread\_fence} function returns no value.

7.17.4.2 The \texttt{atomic\_signal\_fence} function

Synopsis
1

\begin{verbatim}
#include <stdatomic.h>
void atomic_signal_fence(memory_order order);
\end{verbatim}

Description
2 Equivalent to \texttt{atomic\_thread\_fence(order)}, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

3 \textbf{NOTE 1} The \texttt{atomic\_signal\_fence} function can be used to specify the order in which actions performed by the thread become visible to the signal handler.

4 \textbf{NOTE 2} Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with \texttt{atomic\_thread\_fence}, but the hardware fence instructions that \texttt{atomic\_thread\_fence} would have inserted are not emitted.
Returns
5 The `atomic_signal_fence` function returns no value.

7.17.5 Lock-free property
1 The atomic lock-free macros indicate the lock-free property of integer and address atomic types. A value of 0 indicates that the type is never lock-free; a value of 1 indicates that the type is sometimes lock-free; a value of 2 indicates that the type is always lock-free.

Recommended practice
2 Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication via memory mapped into a process more than once and memory shared between two processes.

7.17.5.1 The `atomic_is_lock_free` generic function

Synopsis
1

```c
#include <stdatomic.h>
_Bool atomic_is_lock_free(const volatile A *obj);
```

Description
2 The `atomic_is_lock_free` generic function indicates whether or not atomic operations on objects of the type pointed to by `obj` are lock-free.

Returns
3 The `atomic_is_lock_free` generic function returns nonzero (true) if and only if atomic operations on objects of the type pointed to by the argument are lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.\(^\text{276}\)

7.17.6 Atomic integer types
1 For each line in the following table,\(^\text{277}\) the atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type.\(^\text{278}\)

<table>
<thead>
<tr>
<th>Atomic type name</th>
<th>Direct type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atomic_bool</code></td>
<td><code>_Atomic _Bool</code></td>
</tr>
<tr>
<td><code>atomic_char</code></td>
<td><code>_Atomic char</code></td>
</tr>
<tr>
<td><code>atomic_schar</code></td>
<td><code>_Atomic signed char</code></td>
</tr>
<tr>
<td><code>atomic_uchar</code></td>
<td><code>_Atomic unsigned char</code></td>
</tr>
<tr>
<td><code>atomic_short</code></td>
<td><code>_Atomic short</code></td>
</tr>
<tr>
<td><code>atomic_ushort</code></td>
<td><code>_Atomic unsigned short</code></td>
</tr>
<tr>
<td><code>atomic_int</code></td>
<td><code>_Atomic int</code></td>
</tr>
<tr>
<td><code>atomic_uint</code></td>
<td><code>_Atomic unsigned int</code></td>
</tr>
<tr>
<td><code>atomic_long</code></td>
<td><code>_Atomic long</code></td>
</tr>
<tr>
<td><code>atomic_ulong</code></td>
<td><code>_Atomic unsigned long</code></td>
</tr>
<tr>
<td><code>atomic_ullong</code></td>
<td><code>_Atomic unsigned long long</code></td>
</tr>
<tr>
<td><code>atomic_char16_t</code></td>
<td><code>_Atomic char16_t</code></td>
</tr>
<tr>
<td><code>atomic_char32_t</code></td>
<td><code>_Atomic char32_t</code></td>
</tr>
<tr>
<td><code>atomic_wchar_t</code></td>
<td><code>_Atomic wchar_t</code></td>
</tr>
<tr>
<td><code>atomic_int_least8_t</code></td>
<td><code>_Atomic int_least8_t</code></td>
</tr>
<tr>
<td><code>atomic_uint_least8_t</code></td>
<td><code>_Atomic uint_least8_t</code></td>
</tr>
<tr>
<td><code>atomic_int_least16_t</code></td>
<td><code>_Atomic int_least16_t</code></td>
</tr>
</tbody>
</table>

\(^\text{276}\) `obj` can be a null pointer.

\(^\text{277}\) See “future library directions” (7.31.10).

\(^\text{278}\) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.
Recommended practice

2 The representation of an atomic integer type is not required to have the same size as the corresponding regular type but it should have the same size whenever possible, as it eases effort required to port existing code.

7.17.7 Operations on atomic types

There are only a few kinds of operations on atomic types, though there are many instances of those kinds. This subclause specifies each general kind.

7.17.7.1 The `atomic_store` generic functions

Synopsis

```
#include <stdatomic.h>

void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_acquire`, `memory_order_consume`, nor `memory_order_acq_rel`. Atomically replace the value pointed to by `object` with the value of `desired`. Memory is affected according to the value of `order`.

Returns

3 The `atomic_store` generic functions return no value.

7.17.7.2 The `atomic_load` generic functions

Synopsis

```
#include <stdatomic.h>

C atomic_load(const volatile A *object);
C atomic_load_explicit(const volatile A *object, memory_order order);
```

Description

2 The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. Memory is affected according to the value of `order`. 

§ 7.17.7.2 Library
Returns

3 Atomically returns the value pointed to by `object`.

7.17.7.3 The `atomic_exchange` generic functions

Synopsis

```c
#include <stdatomic.h>

C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
```

Description

2 Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (5.1.2.4).

Returns

3 Atomically returns the value pointed to by `object` immediately before the effects.

7.17.7.4 The `atomic_compare_exchange` generic functions

Synopsis

```c
#include <stdatomic.h>

_Bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected,
C desired, memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected,
C desired, memory_order success, memory_order failure);
```

Description

2 The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the success argument.

3 Atomically, compares the contents of the memory pointed to by `object` for equality with that pointed to by `expected`, and if true, replaces the contents of the memory pointed to by `object` with `desired`, and if false, updates the contents of the memory pointed to by `expected` with that pointed to by `object`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. These operations are atomic read-modify-write operations (5.1.2.4).

4 NOTE 1 For example, the effect of `atomic_compare_exchange_strong` is

```c
if (memcmp(object, expected, sizeof(*object)) == 0)
    memcpy(object, &desired, sizeof(*object));
else
    memcpy(expected, object, sizeof(*object));
```

5 A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.

6 NOTE 2 This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.
EXAMPLE A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```c
exp = atomic_load(&cur);
do {
    des = function(exp);
} while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

Returns

The result of the comparison.

7.17.7.5 The `atomic_fetch` and modify generic functions

The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic integer type. None of these operations is applicable to `atomic_bool`. The key, operator, and computation correspondence is:

<table>
<thead>
<tr>
<th>key</th>
<th>op</th>
<th>computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>sub</td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>bitwise inclusive or</td>
</tr>
<tr>
<td>xor</td>
<td>^</td>
<td>bitwise exclusive or</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
</tbody>
</table>

Synopsis

```c
#include <stdatomic.h>
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
```

Description

Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4). For signed integer types, arithmetic is defined to use two’s complement representation with silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

Returns

Atomically, the value pointed to by `object` immediately before the effects.

NOTE The operation of the `atomic_fetch` and modify generic functions are nearly equivalent to the operation of the corresponding `op=` compound assignment operators. The only differences are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated value of the object, whereas the value returned by the `atomic_fetch` and modify generic functions is the previous value of the atomic object.

7.17.8 Atomic flag type and operations

The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

Operations on an object of type `atomic_flag` shall be lock free.

NOTE Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this document. The remaining types can be emulated with `atomic_flag`, though with less than ideal properties.

The macro `ATOMIC_FLAG_INIT` may be used to initialize an `atomic_flag` to the clear state. An `atomic_flag` that is not explicitly initialized with `ATOMIC_FLAG_INIT` is initially in an indeterminate state.

EXAMPLE
7.17.8.1 The `atomic_flag_test_and_set` functions

Synopsis

```c
#include <stdatomic.h>

_Bool atomic_flag_test_and_set volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit volatile atomic_flag *object,
    memory_order order);
```

Description

Atomically places the atomic flag pointed to by `object` in the set state and returns the value corresponding to the immediately preceding state. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4).

Returns

The `atomic_flag_test_and_set` functions return the value that corresponds to the state of the atomic flag immediately before the effects. The return value true corresponds to the set state and the return value false corresponds to the clear state.

7.17.8.2 The `atomic_flag_clear` functions

Synopsis

```c
#include <stdatomic.h>

void atomic_flag_clear volatile atomic_flag *object);
void atomic_flag_clear_explicit volatile atomic_flag *object,
    memory_order order);
```

Description

The `order` argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`. Atomically places the atomic flag pointed to by `object` into the clear state. Memory is affected according to the value of `order`.

Returns

The `atomic_flag_clear` functions return no value.
7.18 Boolean type and values <stdbool.h>

The header <stdbool.h> defines four macros.

The macro

```
bool
```

expands to _Bool.

The remaining three macros are suitable for use in #if preprocessing directives. They are

```
true
```

which expands to the integer constant 1,

```
false
```

which expands to the integer constant 0, and

```
__bool_true_false_are_defined
```

which expands to the integer constant 1.

Notwithstanding the provisions of 7.1.3, a program may undefine and perhaps then redefine the macros bool, true, and false.\(^{279}\)

\(^{279}\)See “future library directions” (7.31.11).
7.19 Common definitions <stddef.h>

The header <stddef.h> defines the following macros and declares the following types. Some are also defined in other headers, as noted in their respective subclauses.

The types are

- `ptrdiff_t`
  which is the signed integer type of the result of subtracting two pointers;

- `size_t`
  which is the unsigned integer type of the result of the `sizeof` operator;

- `max_align_t`
  which is an object type whose alignment is the greatest fundamental alignment; and

- `wchar_t`
  which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero. Each member of the basic character set shall have a code value equal to its value when used as the lone character in an integer character constant if an implementation does not define `__STDC_MB_MIGHT_NEQ_WC__`.

The macros are

- `NULL`
  which expands to an implementation-defined null pointer constant; and

- `offsetof(type, member-designator)`
  which expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member `subobject` (designated by `member-designator`), from the beginning of its structure (designated by `any object of type type`). The type and member designator shall be such that given

```
static type t;
```

then the expression `&(t, member-designator)` evaluates to an address constant. (If the specified `type` defines a new type or if the specified `member` is a bit-field, the behavior is undefined.)

Recommended practice

The types used for `size_t` and `ptrdiff_t` should not have an integer conversion rank greater than that of `signed long int` unless the implementation supports objects large enough to make this necessary.
7.20 Integer types `<stdint.h>`

The header `<stdint.h>` declares sets of integer types having specified widths, and defines corresponding sets of macros. It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

Types are defined in the following categories:

- integer types having certain exact widths;
- integer types having at least certain specified widths;
- fastest integer types having at least certain specified widths;
- integer types wide enough to hold pointers to objects;
- integer types having greatest width.

(Some of these types may denote the same type.)

Corresponding macros specify limits of the declared types and construct suitable constants.

For each type described herein that the implementation provides, `<stdint.h>` shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, `<stdint.h>` shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as “required”, but need not provide any of the others (described as “optional”).

The feature test macro `__STDC_VERSION_STDINT_H__` expands to the token `yyyymmL`.

7.20.1 Integer types

When typedef names differing only in the absence or presence of the initial `u` are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

In the following descriptions, the symbol `N` represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

7.20.1.1 Exact-width integer types

The typedef name `intN_t` designates a signed integer type with width `N`, no padding bits, and a two’s complement representation. Thus, `int8_t` denotes such a signed integer type with a width of exactly 8 bits.

The typedef name `uintN_t` designates an unsigned integer type with width `N` and no padding bits. Thus, `uint24_t` denotes such an unsigned integer type with a width of exactly 24 bits.

These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, no padding bits, and (for the signed types) that have a two’s complement representation, it shall define the corresponding typedef names.

7.20.1.2 Minimum-width integer types

The typedef name `int_leastN_t` designates a signed integer type with a width of at least `N`, such that no signed integer type with lesser size has at least the specified width. Thus, `int_least32_t` denotes a signed integer type with a width of at least 32 bits.

The typedef name `uint_leastN_t` designates an unsigned integer type with a width of at least `N`, such that no unsigned integer type with lesser size has at least the specified width. Thus, `uint_least16_t` denotes an unsigned integer type with a width of at least 16 bits.

The following types are required:

---

280) See “future library directions” (7.31.12).
281) Some of these types might denote implementation-defined extended integer types.
int_least8_t  uint_least8_t
int_least16_t uint_least16_t
int_least32_t uint_least32_t
int_least64_t uint_least64_t

All other types of this form are optional.

#### 7.20.1.3 Fastest minimum-width integer types

1. Each of the following types designates an integer type that is usually fastest\(^{282}\) to operate with among all integer types that have at least the specified width.

2. The typedef name \texttt{int_fastN\_t} designates the fastest signed integer type with a width of at least \(N\). The typedef name \texttt{uint_fastN\_t} designates the fastest unsigned integer type with a width of at least \(N\).

3. The following types are required:

\[
\begin{array}{ll}
\text{int_fast8\_t} & \text{uint_fast8\_t} \\
\text{int_fast16\_t} & \text{uint_fast16\_t} \\
\text{int_fast32\_t} & \text{uint_fast32\_t} \\
\text{int_fast64\_t} & \text{uint_fast64\_t}
\end{array}
\]

All other types of this form are optional.

#### 7.20.1.4 Integer types capable of holding object pointers

1. The following type designates a signed integer type with the property that any valid pointer to \texttt{void} can be converted to this type, then converted back to pointer to \texttt{void}, and the result will compare equal to the original pointer:

\[
\text{intptr\_t}
\]

The following type designates an unsigned integer type with the property that any valid pointer to \texttt{void} can be converted to this type, then converted back to pointer to \texttt{void}, and the result will compare equal to the original pointer:

\[
\text{uintptr\_t}
\]

These types are optional.

#### 7.20.1.5 Greatest-width integer types

1. The following type designates a signed integer type capable of representing any value of any signed integer type:

\[
\text{intmax\_t}
\]

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

\[
\text{uintmax\_t}
\]

These types are required.

#### 7.20.2 Limits of specified-width integer types

1. The following object-like macros specify the minimum and maximum limits of the types declared in \texttt{<stdint.h>}. Each macro name corresponds to a similar type name in 7.20.1.

\(^{282}\)The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.
Each instance of any defined macro shall be replaced by a constant expression suitable for use in `#if`
preprocessing directives, and, except for the width-of-type macros, this expression shall have the
same type as would an expression that is an object of the corresponding type converted according to
the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude
:absolute value) than the corresponding value given below, with the same sign, except where stated
to be exactly the given value.

### 7.20.2.1 Limits of exact-width integer types

- minimum values of exact-width signed integer types

\[
\text{INT}_N\_MIN \quad \text{exactly} \quad -(2^{N-1})
\]

- maximum values of exact-width signed integer types

\[
\text{INT}_N\_MAX \quad \text{exactly} \quad 2^{N-1} - 1
\]

- maximum values of exact-width unsigned integer types

\[
\text{UINT}_N\_MAX \quad \text{exactly} \quad 2^N - 1
\]

- width of exact-width signed integer types

\[
\text{INT}_N\_WIDTH \quad \text{exactly} \quad N
\]

- width of exact-width unsigned integer types

\[
\text{UINT}_N\_WIDTH \quad \text{exactly} \quad N
\]

### 7.20.2.2 Limits of minimum-width integer types

- minimum values of minimum-width signed integer types

\[
\text{INT}\_\text{LEAST}_N\_MIN \quad -(2^{N-1} - 1)
\]

- maximum values of minimum-width signed integer types

\[
\text{INT}\_\text{LEAST}_N\_MAX \quad 2^{N-1} - 1
\]

- maximum values of minimum-width unsigned integer types

\[
\text{UINT}\_\text{LEAST}_N\_MAX \quad 2^N - 1
\]

- width of minimum-width signed integer types

\[
\text{INT}\_\text{LEAST}_N\_WIDTH \quad N
\]

- width of minimum-width unsigned integer types

\[
\text{UINT}\_\text{LEAST}_N\_WIDTH \quad N
\]
7.20.2.3 Limits of fastest minimum-width integer types

- Minimum values of fastest minimum-width signed integer types
  \[
  \text{INT\_FAST}_N\_MIN = -(2^{N-1} - 1)
  \]

- Maximum values of fastest minimum-width signed integer types
  \[
  \text{INT\_FAST}_N\_MAX = 2^{N-1} - 1
  \]

- Maximum values of fastest minimum-width unsigned integer types
  \[
  \text{UINT\_FAST}_N\_MAX = 2^N - 1
  \]

- Width of fastest minimum-width signed integer types
  \[
  \text{INT\_FAST}_N\_WIDTH = N
  \]

- Width of fastest minimum-width unsigned integer types
  \[
  \text{UINT\_FAST}_N\_WIDTH = N
  \]

7.20.2.4 Limits of integer types capable of holding object pointers

- Minimum value of pointer-holding signed integer type
  \[
  \text{INTPTR\_MIN} = -(2^{15} - 1)
  \]

- Maximum value of pointer-holding signed integer type
  \[
  \text{INTPTR\_MAX} = 2^{15} - 1
  \]

- Maximum value of pointer-holding unsigned integer type
  \[
  \text{UINTPTR\_MAX} = 2^{16} - 1
  \]

- Width of pointer-holding signed integer type
  \[
  \text{INTPTR\_WIDTH} = 16
  \]

- Width of pointer-holding unsigned integer type
  \[
  \text{UINTPTR\_WIDTH} = 16
  \]

7.20.2.5 Limits of greatest-width integer types

- Minimum value of greatest-width signed integer type
  \[
  \text{INTMAX\_MIN} = -(2^{63} - 1)
  \]

- Maximum value of greatest-width signed integer type
  \[
  \text{INTMAX\_MAX} = 2^{63} - 1
  \]
maximum value of greatest-width unsigned integer type

\[ \text{UINTMAX\_MAX} = 2^{64} - 1 \]

width of greatest-width signed integer type

\[ \text{INTMAX\_WIDTH} = 64 \]

width of greatest-width unsigned integer type

\[ \text{UINTMAX\_WIDTH} = 64 \]

### 7.20.3 Limits of other integer types

1. The following object-like macros specify the minimum and maximum limits of integer types corresponding to types defined in other standard headers.
2. Each instance of these macros shall be replaced by a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type, except for the width-of-type macros, as would an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign. An implementation shall define only the macros corresponding to those typedef names it actually provides.\(^{283}\)

- limits of `ptrdiff_t`

\[
\begin{align*}
\text{PTDIFF\_MIN} &= -65535 \\
\text{PTDIFF\_MAX} &= +65535 \\
\text{PTDIFF\_WIDTH} &= 16 \\
\end{align*}
\]

- limits of `sig_atomic_t`

\[
\begin{align*}
\text{SIG\_ATOMIC\_MIN} &= \text{see below} \\
\text{SIG\_ATOMIC\_MAX} &= \text{see below} \\
\text{SIG\_ATOMIC\_WIDTH} &= 8 \\
\end{align*}
\]

- limit of `size_t`

\[
\begin{align*}
\text{SIZE\_MAX} &= 65535 \\
\text{SIZE\_WIDTH} &= 16 \\
\end{align*}
\]

- limits of `wchar_t`

\[
\begin{align*}
\text{WCHAR\_MIN} &= \text{see below} \\
\text{WCHAR\_MAX} &= \text{see below} \\
\text{WCHAR\_WIDTH} &= 8 \\
\end{align*}
\]

- limits of `wint_t`

\[
\begin{align*}
\text{WINT\_MIN} &= \text{see below} \\
\text{WINT\_MAX} &= \text{see below} \\
\text{WINT\_WIDTH} &= 16 \\
\end{align*}
\]

\(^{283}\)A freestanding implementation need not provide all of these types.
If `sig_atomic_t` (see 7.14) is defined as a signed integer type, the value of `SIG_ATOMIC_MIN` shall be no greater than \(-127\) and the value of `SIG_ATOMIC_MAX` shall be no less than 127; otherwise, `sig_atomic_t` is defined as an unsigned integer type, and the value of `SIG_ATOMIC_MIN` shall be 0 and the value of `SIG_ATOMIC_MAX` shall be no less than 255.  

If `wchar_t` (see 7.19) is defined as a signed integer type, the value of `WCHAR_MIN` shall be no greater than \(-127\) and the value of `WCHAR_MAX` shall be no less than 127; otherwise, `wchar_t` is defined as an unsigned integer type, and the value of `WCHAR_MIN` shall be 0 and the value of `WCHAR_MAX` shall be no less than 255. \(^{284}\)  

If `wint_t` (see 7.29) is defined as a signed integer type, the value of `WINT_MIN` shall be no greater than \(-32767\) and the value of `WINT_MAX` shall be no less than 32767; otherwise, `wint_t` is defined as an unsigned integer type, and the value of `WINT_MIN` shall be 0 and the value of `WINT_MAX` shall be no less than 65535.

7.20.4 Macros for integer constants

The following function-like macros expand to integer constant expressions suitable for initializing objects that have integer types corresponding to types defined in `<stdint.h>`. Each macro name corresponds to a similar type name in 7.20.1.2 or 7.20.1.5.  

The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.  

Each invocation of one of these macros shall expand to an integer constant expression suitable for use in `#if` preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

7.20.4.1 Macros for minimum-width integer constants

The macro `INTN_C(value)` expands to an integer constant expression corresponding to the type `int_leastN_t`. The macro `UINTN_C(value)` expands to an integer constant expression corresponding to the type `uint_leastN_t`. For example, if `uint_least64_t` is a name for the type `unsigned long long int`, then `UINT64_C(0x123)` might expand to the integer constant `0x123ULL`.  

7.20.4.2 Macros for greatest-width integer constants

The following macro expands to an integer constant expression having the value specified by its argument and the type `intmax_t`:

```
INTMAX_C(value)
```

The following macro expands to an integer constant expression having the value specified by its argument and the type `uintmax_t`:

```
UINTMAX_C(value)
```

\(^{284}\)The values `WCHAR_MIN` and `WCHAR_MAX` do not necessarily correspond to members of the extended character set.
7.21 Input/output \texttt{<stdio.h>}

7.21.1 Introduction

The header \texttt{<stdio.h>} defines several macros, and declares three types and many functions for performing input and output.

The types declared are \texttt{size_t} (described in 7.19);

\begin{verbatim}
FILE
\end{verbatim}

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an \texttt{error indicator} that records whether a read/write error has occurred, and an \texttt{end-of-file indicator} that records whether the end of the file has been reached; and

\begin{verbatim}
fpos_t
\end{verbatim}

which is a complete object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

The macros are \texttt{NULL} (described in 7.19);

\begin{verbatim}
_IOFBF
_IOLBF
_INOBF
\end{verbatim}

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the \texttt{setvbuf} function;

\begin{verbatim}
BUFSIZ
\end{verbatim}

which expands to an integer constant expression that is the size of the buffer used by the \texttt{setbuf} function;

\begin{verbatim}
EOF
\end{verbatim}

which expands to an integer constant expression, with type \texttt{int} and a negative value, that is returned by several functions to indicate end-of-file, that is, no more input from a stream;

\begin{verbatim}
FOPEN_MAX
\end{verbatim}

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

\begin{verbatim}
FILENAME_MAX
\end{verbatim}

which expands to an integer constant expression that is the size needed for an array of \texttt{char} large enough to hold the longest file name string that the implementation guarantees can be opened or, if the implementation imposes no practical limit on the length of file name strings, the recommended size of an array intended to hold a file name string;\footnote{Of course, file name string contents are subject to other system-specific constraints; therefore all possible strings of length \texttt{FILENAME_MAX} cannot be expected to be opened successfully.}

\begin{verbatim}
L_tmpnam
\end{verbatim}

which expands to an integer constant expression that is the size needed for an array of \texttt{char} large enough to hold a temporary file name string generated by the \texttt{tmpnam} function;
which expand to integer constant expressions with distinct values, suitable for use as the third argument to the `fseek` function;

which expands to an integer constant expression that is the minimum number of unique file names that can be generated by the `tmpnam` function;

which are expressions of type “pointer to FILE” that point to the FILE objects associated, respectively, with the standard error, input, and output streams.

4 The header `<wchar.h>` declares a number of functions useful for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of “generalized” multibyte characters, as described further in 7.21.3.

5 The input/output functions are given the following collective terms:

- The wide character input functions — those functions described in 7.29 that perform input into wide characters and wide strings: `fgetwc`, `fgetws`, `getwc`, `getwchar`, `fwscanf`, `wscanf`, `vfwscanf`, and `vwscanf`.

- The wide character output functions — those functions described in 7.29 that perform output from wide characters and wide strings: `fputwc`, `fputws`, `putwc`, `putwchar`, `fwprintf`, `wprintf`, `vfwprintf`, and `vwprintf`.

- The wide character input/output functions — the union of the ungetwc function, the wide character input functions, and the wide character output functions.

- The byte input/output functions — those functions described in this subclause that perform input/output: `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `getc`, `getchar`, `printf`, `putc`, `putchar`, `puts`, `scanf`, `ungetc`, `vfprintf`, `vfscanf`, `vprintf`, and `vscanf`.

Forward references: files (7.21.3), the `fseek` function (7.21.9.2), streams (7.21.2), the `tmpnam` function (7.21.4.4), `<wchar.h>` (7.29).

7.21.2 Streams

1 Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for text streams and for binary streams. 286

2 A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a

286 An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.
stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

3 A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

4 Each stream has an orientation. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the stream becomes a wide-oriented stream. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a byte-oriented stream. Only a call to the freopen function or the fwide function can otherwise alter the orientation of a stream. (A successful call to freopen removes any orientation.)

5 Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream’s orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

6 Each wide-oriented stream has an associated mbstate_t object that stores the current parse state of the stream. A successful call to fgetpos stores a representation of the value of this mbstate_t object as part of the value of the fpos_t object. A later successful call to fsetpos using the same stored fpos_t value restores the value of the associated mbstate_t object as well as the position within the controlled stream.

7 Each stream has an associated lock that is used to prevent data races when multiple threads of execution access a stream, and to restrict the interleaving of stream operations performed by multiple threads. Only one thread may hold this lock at a time. The lock is reentrant: a single thread may hold the lock multiple times at a given time.

8 All functions that read, write, position, or query the position of a stream lock the stream before accessing it. They release the lock associated with the stream when the access is complete.

Environmental limits

9 An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro BUFSIZE shall be at least 256.

Forward references: the freopen function (7.21.5.4 ), the fwide function (7.29.3.5), mbstate_t (7.29.1), the fgetpos function (7.21.9.1 ), the fsetpos function (7.21.9.3 ).

7.21.3 Files

1 A stream is associated with an external file (which may be a physical device) by opening a file, which may involve creating a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a file position indicator associated with the stream is positioned at the start (character number
zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

2 Binary files are not truncated, except as defined in 7.21.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

3 When a stream is unbuffered, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is fully buffered, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is line buffered, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the `setbuf` and `setvbuf` functions.

4 A file may be disassociated from a controlling stream by closing the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

5 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the main function returns to its original caller, or if the `exit` function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the `abort` function, need not close all files properly.

6 The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object need not serve in place of the original.

7 At program startup, three text streams are predefined and need not be opened explicitly — standard input (for reading conventional input), standard output (for writing conventional output), and standard error (for writing diagnostic output). As initially opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

8 Functions that open additional (nontemporary) files require a file name, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

9 Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

   - Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).
   - A file need not begin nor end in the initial shift state.\(^{288}\)

10 Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

11 The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the `fgetwc` function. Each conversion occurs as if by a call to the `mbtowc` function, with the conversion state described by the stream's

\(^{288}\) Setting the file position indicator to end-of-file, as with `fseek(file, 0, SEEK_END)`, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.
own `mbstate_t` object. The byte input functions read characters from the stream as if by successive calls to the `fgetc` function.

The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the `fputwc` function. Each conversion occurs as if by a call to the `wcrtomb` function, with the conversion state described by the stream’s own `mbstate_t` object. The byte output functions write characters to the stream as if by successive calls to the `fputc` function.

In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the `mbtowc` and `wcrtomb` functions.

An encoding error occurs if the character sequence presented to the underlying `mbtowc` function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying `wcrtomb` does not correspond to a valid (generalized) multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro `EILSEQ` in `errno` if and only if an encoding error occurs.

Environmental limits

The value of `FOPEN_MAX` shall be at least eight, including the three standard text streams.

Forward references: the `exit` function (7.22.4.4), the `fgetc` function (7.21.7.1), the `fopen` function (7.21.5.3), the `fputc` function (7.21.7.3), the `setbuf` function (7.21.5.5), the `setvbuf` function (7.21.5.6), the `fgetwc` function (7.29.3.1), the `fputwc` function (7.29.3.3), conversion state (7.29.6), the `mbtowc` function (7.29.6.3.2), the `wcrtomb` function (7.29.6.3.3).

7.21.4 Operations on files

7.21.4.1 The `remove` function

Synopsis

```c
#include <stdio.h>
int remove(const char *filename);
```

Description

The `remove` function causes the file whose name is the string pointed to by `filename` to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the `remove` function is implementation-defined.

Returns

The `remove` function returns zero if the operation succeeds, nonzero if it fails.

7.21.4.2 The `rename` function

Synopsis

```c
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

The `rename` function causes the file whose name is the string pointed to by `old` to be henceforth known by the name given by the string pointed to by `new`. The file named `old` is no longer accessible by that name. If a file named by the string pointed to by `new` exists prior to the call to the `rename` function, the behavior is implementation-defined.

Returns

The `rename` function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

Among the reasons the implementation could cause the `rename` function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.
7.21.4.3 The tmpfile function

Synopsis

#include <stdio.h>
FILE *tmpfile(void);

Description

The tmpfile function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode.

Recommended practice

It should be possible to open at least TMP_MAX temporary files during the lifetime of the program (this limit may be shared with tmpnam) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (FOPEN_MAX).

Returns

The tmpfile function returns a pointer to the stream of the file that it created. If the file cannot be created, the tmpfile function returns a null pointer.

Forward references: the fopen function (7.21.5.3).

7.21.4.4 The tmpnam function

Synopsis

#include <stdio.h>
char *tmpnam(char *s);

Description

The tmpnam function generates a string that is a valid file name and that is not the same as the name of an existing file. The function is potentially capable of generating at least TMP_MAX different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

The tmpnam function generates a different string each time it is called.

Calls to the tmpnam function with a null pointer argument may introduce data races with each other. The implementation shall behave as if no library function calls the tmpnam function.

Returns

If no suitable string can be generated, the tmpnam function returns a null pointer. Otherwise, if the argument is a null pointer, the tmpnam function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the tmpnam function may modify the same object). If the argument is not a null pointer, it is assumed to point to an array of at least L_tmpnam chars; the tmpnam function writes its result in that array and returns the argument as its value.

Environmental limits

The value of the macro TMP_MAX shall be at least 25.

7.21.5 File access functions

7.21.5.1 The fclose function

Synopsis

#include <stdio.h>
int fclose(FILE *stream);

Files created using strings generated by the tmpnam function are temporary only in the sense that their names are not expected to collide with those generated by conventional naming rules for the implementation. It is still necessary to use the remove function to remove such files when their use is ended, and before program termination.
Description

A successful call to the `fclose` function causes the stream pointed to by `stream` to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether or not the call succeeds, the stream is disassociated from the file and any buffer set by the `setbuf` or `setvbuf` function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

The `fclose` function returns zero if the stream was successfully closed, or `EOF` if any errors were detected.

7.21.5.2 The `fflush` function

Synopsis

```c
#include <stdio.h>
int fflush(FILE *stream);
```

Description

If `stream` points to an output stream or an update stream in which the most recent operation was not input, the `fflush` function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

If `stream` is a null pointer, the `fflush` function performs this flushing action on all streams for which the behavior is defined above.

Returns

The `fflush` function sets the error indicator for the stream and returns `EOF` if a write error occurs, otherwise it returns zero.

Forward references: the `fopen` function (7.21.5.3).

7.21.5.3 The `fopen` function

Synopsis

```c
#include <stdio.h>
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

Description

The `fopen` function opens the file whose name is the string pointed to by `filename`, and associates a stream with it.

The argument `mode` points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.\(^{201}\)

- `r`: open text file for reading
- `w`: truncate to zero length or create text file for writing
- `wx`: create text file for writing
- `a`: append; open or create text file for writing at end-of-file
- `rb`: open binary file for reading
- `wb`: truncate to zero length or create binary file for writing
- `wbx`: create binary file for writing
- `ab`: append; open or create binary file for writing at end-of-file
- `r+`: open text file for update (reading and writing)

\(^{201}\)If the string begins with one of the above sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in 7.21.2).
w+  truncate to zero length or create text file for update
w+x  create text file for update
a+  append; open or create text file for update, writing at end-of-file
r+b or rb+  open binary file for update (reading and writing)
w+b or wb+  truncate to zero length or create binary file for update
w+bx or wb+x  create binary file for update
a+b or ab+  append; open or create binary file for update, writing at end-of-file

4 Opening a file with read mode (‘r’ as the first character in the mode argument) fails if the file does not exist or cannot be read.
5 Opening a file with exclusive mode (‘x’ as the last character in the mode argument) fails if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non-shared) access to the extent that the underlying system supports exclusive access.
6 Opening a file with append mode (‘a’ as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the fseek function. In some implementations, opening a binary file with append mode (‘b’ as the second or third character in the above list of mode argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.
7 When a file is opened with update mode (‘+’ as the second or third character in the above list of mode argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the fflush function or to a file positioning function (fseek, fsetpos, or rewind), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.
8 When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns
9 The fopen function returns a pointer to the object controlling the stream. If the open operation fails, fopen returns a null pointer.

Forward references:  file positioning functions (7.21.9 ).

7.21.5.4 The freopen function

Synopsis

```c
#include <stdio.h>
FILE *freopen(const char * restrict filename, const char * restrict mode, FILE * restrict stream);
```

Description
2 The freopen function opens the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument is used just as in the fopen function.²⁹²
3 If filename is a null pointer, the freopen function attempts to change the mode of the stream to that specified by mode, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.
4 The freopen function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

²⁹² The primary use of the freopen function is to change the file associated with a standard text stream (stderr, stdin, or stdout), as those identifiers need not be modifiable lvalues to which the value returned by the fopen function could be assigned.
Returns

The `freopen` function returns a null pointer if the open operation fails. Otherwise, `freopen` returns the value of `stream`.

### 7.21.5.5 The `setbuf` function

#### Synopsis

```c
#include <stdio.h>
void setbuf(FILE * restrict stream, char * restrict buf);
```

#### Description

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for `mode` and `BUFSIZ` for `size`, or (if `buf` is a null pointer), with the value `_IONBF` for `mode`.

#### Returns

The `setbuf` function returns no value.

**Forward references:** the `setvbuf` function (7.21.5.6).

### 7.21.5.6 The `setvbuf` function

#### Synopsis

```c
#include <stdio.h>
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
```

#### Description

The `setvbuf` function may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation (other than an unsuccessful call to `setvbuf`) is performed on the stream. The argument `mode` determines how `stream` will be buffered, as follows:

- `_IOFBF` causes input/output to be fully buffered;
- `_IOLBF` causes input/output to be line buffered;
- `_IONBF` causes input/output to be unbuffered.

If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by the `setvbuf` function and the argument `size` specifies the size of the array; otherwise, `size` may determine the size of a buffer allocated by the `setvbuf` function. The contents of the array at any time are indeterminate.

#### Returns

The `setvbuf` function returns zero on success, or nonzero if an invalid value is given for `mode` or if the request cannot be honored.

### 7.21.6 Formatted input/output functions

The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.

#### 7.21.6.1 The `fprintf` function

##### Synopsis

```c
#include <stdio.h>
int fprintf(FILE * restrict stream, const char * restrict format, ...);
```

---

293) The buffer has to have a lifetime at least as great as the open stream, so not closing the stream before a buffer that has automatic storage duration is deallocated upon block exit results in undefined behavior.

294) The `fprintf` functions perform writes to memory for the `%n` specifier.
Description

2 The `fprintf` function writes output to the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

3 The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

4 Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

   — Zero or more flags (in any order) that modify the meaning of the conversion specification.
   — An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a nonnegative decimal integer.295
   — An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point character for a, A, e, E, f, and F conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of bytes to be written for s conversions. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
   — An optional length modifier that specifies the size of the argument.
   — A conversion specifier character that specifies the type of conversion to be applied.

5 As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

6 The flag characters and their meanings are:

   - The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
   + The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)296
   space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.
   # The result is converted to an “alternative form”. For o conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For x (or X) conversion, a nonzero result has 0x (or 0X) prefixed to it. For a, A, e, E, f, F, g, and G conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a

295 Note that 0 is taken as a flag, not as the beginning of a field width.
296 The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

276 Library § 7.21.6.1
decimal-point character appears in the result of these conversions only if a digit follows it.)
For g and G conversions, trailing zeros are not removed from the result. For other conversions,
the behavior is undefined.

For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversions, leading zeros (following any indication
of sign or base) are used to pad to the field width rather than performing space padding,
except when converting an infinity or NaN. If the θ and - flags both appear, the θ flag is
ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the θ flag is ignored.
For other conversions, the behavior is undefined.

7 The length modifiers and their meanings are:

hh Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char
or unsigned char argument (the argument will have been promoted according to the
integer promotions, but its value shall be converted to signed char or unsigned char
before printing); or that a following n conversion specifier applies to a pointer to a
signed char argument.

h Specifies that a following d, i, o, u, x, or X conversion specifier applies to a short int
or unsigned short int argument (the argument will have been promoted according to the
integer promotions, but its value shall be converted to short int or unsigned short int
before printing); or that a following n conversion specifier applies to a pointer to a
short int argument.

l (ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a long int
or unsigned long int argument; that a following n conversion specifier applies to a
pointer to a long int argument; that a following c conversion specifier applies to a
wint_t argument; that a following s conversion specifier applies to a pointer to a
wchar_t argument; or has no effect on a following a, A, e, E, f, F, g, or G conversion
specifier.

ll (ell-ell) Specifies that a following d, i, o, u, x, or X conversion specifier applies to a
long long int or unsigned long long int argument; or that a following n conversion
specifier applies to a pointer to a long long int argument.

j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax_t
or uintmax_t argument; or that a following n conversion specifier applies to a pointer
to an intmax_t argument.

z Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size_t or the
corresponding signed integer type argument; or that a following n conversion
specifier applies to a pointer to a signed integer type corresponding to size_t argument.

t Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff_t
or the corresponding unsigned integer type argument; or that a following n conversion
specifier applies to a pointer to a ptrdiff_t argument.

L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a
long double argument.

H Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a
_decimal32 argument.

D Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a
_decimal64 argument.

DD Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to a
_decimal128 argument.
If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

- **d, i** The int argument is converted to signed decimal in the style [-]dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

- **o, u, x, X** The unsigned int argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X) in the style dddd; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

- **f, F** A double argument representing a floating-point number is converted to decimal notation in the style [-]dddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. The result of converting a zero value with a precision of zero is no characters.

- **e, E** A double argument representing an infinity is converted in one of the styles [-]inf or [-]infinity — which style is implementation-defined. A double argument representing a NaN is converted in one of the styles [-]nan or [-]nan(n-char-sequence) — which style, and the meaning of any n-char-sequence, is implementation-defined. The F conversion specifier produces INF, INFINITY, or NAN instead of inf, infinity, or nan, respectively.\(^{297}\)

- **g, G** A double argument representing a floating-point number is converted in style [-]ddddddd.ddd, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

- **a, A** A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

---

\(^{297}\)When applied to infinite and NaN values, the -, +, and space flag characters have their usual meaning: the # and 0 flag characters have no effect.
character\(^{298}\) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and \texttt{FLT\_RADIX} is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and \texttt{FLT\_RADIX} is not a power of 2, then the precision is sufficient to distinguish\(^{299}\) values of type \texttt{double}, except that trailing zeros may be omitted; if the precision is zero and the \# flag is not specified, no decimal-point character appears. The letters \texttt{abcdef} are used for a conversion and the letters \texttt{ABCDEFG} for a \texttt{A} conversion. The \texttt{A} conversion specifier produces a number with \texttt{X} and \texttt{P} instead of \texttt{x} and \texttt{p}. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A \texttt{double} argument representing an infinity or NaN is converted in the style of an \texttt{f} or \texttt{F} conversion specifier.

If an \texttt{H}, \texttt{D}, or \texttt{DD} modifier is present and the precision is missing, then for a decimal floating type argument represented by a triple of integers \((s,c,q)\), where \(n\) is the number of significant digits in the coefficient \(c\),

- if \((- (n + 5)) \leq q \leq 0\), use style \texttt{f} (or style \texttt{F} in the case of an \texttt{A} conversion specifier) with formatting precision equal to \(- q\).
- otherwise, use style \texttt{e} (or style \texttt{E} in the case of an \texttt{A} conversion specifier) with formatting precision equal to \(n - 1\), with the exceptions that if \(c = 0\) then the digit-sequence in the exponent-part shall have the value \(q\) (rather than \(0\)), and that the exponent is always expressed with the minimum number of digits required to represent its value (the exponent never contains a leading zero).

If the precision \(P\) is present (in the conversion specification) and is zero or at least as large as the precision \(p\) (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision \(P\) is present (and nonzero) and less than the precision \(p\) of the decimal floating type, the conversion first obtains an intermediate result as follows, where \(n\) is the number of significant digits in the coefficient:

- If \(n \leq P\), set the intermediate result to the input.
- If \(n > P\), round the input value, according to the current rounding direction for decimal floating-point operations, to \(P\) decimal digits, with unbounded exponent range, representing the result with a \(P\)-digit integer coefficient when in the form \((s,c,q)\).

Convert the intermediate result in the manner described above for the case where the precision is missing.

\texttt{c}\ If no \texttt{l} length modifier is present, the \texttt{int} argument is converted to an \texttt{unsigned char}, and the resulting character is written.

If an \texttt{l} length modifier is present, the \texttt{wint\_t} argument is converted as if by an \texttt{ls} conversion specification with no precision and an argument that points to the initial element of a two-element array of \texttt{wchar\_t}, the first element containing the \texttt{wint\_t} argument to the \texttt{lc} conversion specification and the second a null wide character.

\texttt{s}\ If no \texttt{l} length modifier is present, the argument shall be a pointer to the initial element of an array of character type.\(^{300}\) Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

\(^{298}\)Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

\(^{299}\)The precision \(p\) is sufficient to distinguish values of the source type if \(16^{n-1} > b^n\) where \(b\) is \texttt{FLT\_RADIX} and \(n\) is the number of base-\(b\) digits in the significand of the source type. A smaller \(p\) might suffice depending on the implementation’s scheme for determining the digit to the left of the decimal-point character.

\(^{300}\)No special provisions are made for multibyte characters.
If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of `wchar_t` type. Wide characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.301)

The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

The argument shall be a pointer to signed integer into which is written the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.302) If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For `a` and `A` conversions, if `FLT_RADIX` is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

For `a` and `A` conversions, if `FLT_RADIX` is not a power of 2 and the result is not exactly representable in the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most the maximum value of the `T_DECIMAL_DIG` macros (defined in `<float.h>`), then the result should be correctly rounded.303) If the number of significant decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings `L < U`, both having `DECIMAL_DIG` significant digits; the value of the resultant decimal string `D` should satisfy `L ≤ D ≤ U`, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

The `fprintf` function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

The number of characters that can be produced by any single conversion shall be at least 4095.

EXAMPLE 1 To print a date and time in the form “Sunday, July 3, 10:02” followed by \( \pi \) to five decimal places:

```c
#include <math.h>
```

301) Redundant shift sequences can result if multibyte characters have a state-dependent encoding.
302) See “future library directions” (7.31.13).
303) For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.
#include <stdio.h>
/* ...
char *weekday, *month;  // pointers to strings
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
    weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));

EXAMPLE 2 In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a and the second by an uppercase letter.

Given the following wide string with length seven,

static wchar_t wstr[] = L"XabcZW";

the seven calls

    fprintf(stdout, "|1234567890123|\n");
    fprintf(stdout, "|[XabcZ|\n");
    fprintf(stdout, "|[XabcZ|\n");
    fprintf(stdout, "|abcZ|\n");
    fprintf(stdout, "|abcZ|\n");
    fprintf(stdout, "|abcZ|\n");
    fprintf(stdout, "|abcZ|\n");

will print the following seven lines:

  |1234567890123|
  □XabcZW
  □XabcZ
  □XabcZ
  □XabcZ
  □XabcZ
  □XabcZ

EXAMPLE 3 Following are representations of _Decimal64 arguments as triples (s, c, q) and the corresponding character sequences fprintf produces with "%Da":

(1, 123, 0)       123
(−1, 123, 0)     -123
(1, 123, −2)      1.23
(−1, 123, −2)   -1.23e+3
(1, 123, −8)    0.00000123
(−1, 123, −8) -0.00000123
(1, 120, −9)    1.23e-7
(−1, 120, −9)   -1.23e-7
(1, 1234567890123456, 0) 1234567890123456
(1, 1234567890123456, 1) 1.234567890123456e+16
(1, 1234567890123456, −1) 123456789012345.6
(1, 1234567890123456, −21) 0.000001234567890123456
(1, 1234567890123456, −22) 1.234567890123456e-7
(1, 0, 0)        0
(−1, 0, 0)       -0
(1, 0, −6)       0.00000
(1, 0, −7)       0e-7
(1, 0, 2)        0e+2
(1, 5, −6)       0.000005
(1, 5, −7)       0.0000050
(1, 5, −7)       5e-7

To illustrate the effects of a precision specification, the sequence:

    _Decimal32 x = 6543.00DF;  // (1, 654300, -2)
assuming default rounding, results in:

6543.00
6543.00
6543.0
6543
6.54e+3
6.5e+3
7e+3
6543.00

To illustrate the effects of the exponent range, the sequence:

```
_decimal32 x = 9543210e87DF; // (1, 9543210, 87)
_decimal32 y = 9500000e90DF; // (1, 9500000, 90)
fprintf(stdout, "%.6Ha\n", x);
fprintf(stdout, "%.5Ha\n", x);
fprintf(stdout, "%.4Ha\n", x);
fprintf(stdout, "%.3Ha\n", x);
fprintf(stdout, "%.2Ha\n", x);
fprintf(stdout, "%.1Ha\n", x);
fprintf(stdout, "%.0Ha\n", x);
```

assuming default rounding, results in:

9.54321e+93
9.5432e+93
9.543e+93
9.54e+93
9e+93
1e+94
1e+97

To further illustrate the effects of the exponent range, the sequence:

```
_decimal32 x = 9512345e90DF; // (1, 9512345, 90)
_decimal32 y = 9512345e86DF; // (1, 9512345, 86)
fprintf(stdout, "%.3Ha\n", x);
fprintf(stdout, "%.2Ha\n", x);
fprintf(stdout, "%.1Ha\n", x);
fprintf(stdout, "%.2Ha\n", y);
```

assuming default rounding, results in:

9.51e+96
9.5e+96
1e+97
9.5e+92

Forward references: conversion state (7.29.6), the `wcrtomb` function (7.29.6.3.3).

7.21.6.2 The fscanf function

**Synopsis replace**

```
#include <stdio.h>

int fscanf(FILE * restrict stream, const char * restrict format, ...);
```
Description

The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional decimal integer greater than zero that specifies the maximum field width (in characters).
- An optional `length modifier` that specifies the size of the receiving object.
- A `conversion specifier` character that specifies the type of conversion to be applied.

The `fscanf` function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. The directive never fails.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

1. Input white-space characters (as specified by the function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier.\(^{204}\)
2. An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.\(^{205}\) The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.
3. Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

\(^{204}\) These white-space characters are not counted against a specified field width.

\(^{205}\) `fscanf` pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to `strtol`, `strtof`, etc., are unacceptable to `fscanf`. 

§ 7.21.6.2 Library 283
hh Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _signed_char or _unsigned_char.

h Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _short_int or _unsigned_short_int.

l (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _long_int or _unsigned_long_int; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _double; or that a following c, s, or l conversion specifier applies to an argument with type pointer to _wchar_t.

ll (ell-ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _long_long_int or _unsigned_long_long_int.

j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _intmax_t or _uintmax_t.

z Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _size_t or the corresponding signed integer type.

t Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to _ptrdiff_t or the corresponding unsigned integer type.

L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _long_double.

H Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal32.

D Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal64.

DD Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal128.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

12 The conversion specifiers and their meanings are:

d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the _strtol function with the value 10 for the base argument. The corresponding argument shall be a pointer to signed integer.

i Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the _strtol function with the value 0 for the base argument. The corresponding argument shall be a pointer to signed integer.

o Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the _strtol function with the value 8 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

u Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the _strtol function with the value 10 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

x Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the _strtol function with the value 16 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

---

ISO/IEC 9899:202x (E)   working draft — November 18, 2019 C17..C2x N2455

284 Library § 7.21.6.2
Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the `strtod` function. The corresponding argument shall be a pointer to floating.

Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive). If no length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the `mbtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the resulting sequence of wide characters. No null wide character is added.

Matches a sequence of non-white-space characters. If no length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the `mbtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Matches a nonempty sequence of characters from a set of expected characters (the scanlist). If no length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the `mbtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket (`). The characters between the brackets (the scanlist) compose the scanlist, unless the character after the left bracket is a circumflex (`^`), in which case the scanlist contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

---

No special provisions are made for multibyte characters in the matching rules used by the `c`, `s`, and `[]` conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

§ 7.21.6.2 Library 285
Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the `fprintf` function. The corresponding argument shall be a pointer to a pointer to `void`. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.

No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the `fscanf` function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the `fscanf` function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

% Matches a single % character; no conversion or assignment occurs. The complete conversion specification shall be ``.

If a conversion specification is invalid, the behavior is undefined.\(^{307}\)

The conversion specifiers A, E, F, G, and X are also valid and behave the same as, respectively, a, e, f, g, and x.

Trailing white-space characters (including new-line characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

Returns

The `fscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

**EXAMPLE 1** The call:

```c
#include <stdio.h>
/* .... */
int n, i; float x; char name[50];
   n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to \(n\) the value 3, to \(i\) the value 25, to \(x\) the value 5.432, and to \(name\) the sequence `thompson\0`.

**EXAMPLE 2** The call:

```c
#include <stdio.h>
/* .... */
int i; float x; char name[50];
   fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to \(i\) the value 56 and to \(x\) the value 789.0, will skip 0123, and will assign to \(name\) the sequence `56\0`. The next character read from the input stream will be `a`.

**EXAMPLE 3** To accept repeatedly from `stdin` a quantity, a unit of measure, and an item name:

```c
#include <stdio.h>
```

See “future library directions” (7.31.13).
20 If the stdin stream contains the following lines:

2 quarts of oil
-12.8 degrees Celsius
lots of luck
10.0LBS of dirt
100ergs of energy

the execution of the above example will be analogous to the following assignments:

```c
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

21 EXAMPLE 4 In:
```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to d1 and the value 3 to n1. Because %n can never get an input failure, the value of 3 is also assigned to n2. The value of d2 is not affected. The value 1 is assigned to i.

22 EXAMPLE 5 The call:
```
#include <stdio.h>
/* ... */
int n, i;
n = sscanf("foo %bar 42", "foo%%bar%d", &i);
```

will assign to n the value 1 and to i the value 42 because input white-space characters are skipped for both the % and d conversion specifiers.

23 EXAMPLE 6 In these examples, multibyte characters do have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a \ Pend and the second by an uppercase letter, but are only recognized as such when in the alternate shift state. The shift sequences are denoted by a Pend \ Pend and i, in which the first causes entry into the alternate shift state.

24 After the call:
```
#include <stdio.h>
/* ... */
char str[50];
fscanf(stdin, "a%s", str);
```

with the input line:
```
 a\ Pend \ Pend bc
```

str will contain a Pend \ Pend \ Pend bc assuming that none of the bytes of the shift sequences (or of the multibyte characters, in the more general case) appears to be a single-byte white-space character.

§7.21.6.2 Library
In contrast, after the call:

```c
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a%ls", wstr);
```

with the same input line, `wstr` will contain the two wide characters that correspond to \u1eaf \u1eaf \u03c0 \u03c0 and \u1ee2 \u1ee2 □ □ and a terminating null wide character.

However, the call:

```c
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a \u2026 \u2026 %ls", wstr);
```

with the same input line will return zero due to a matching failure against the \u2026 \u2026 sequence in the format string.

Assuming that the first byte of the multibyte character \u1eaf \u03c0 is the same as the first byte of the multibyte character \u1ee2 \u03c0, after the call:

```c
#include <stdio.h>
#include <stddef.h>
/* ... */
wchar_t wstr[50];
fscanf(stdin, "a\u03c0\%ls", wstr);
```

with the same input line, zero will again be returned, but `stdin` will be left with a partially consumed multibyte character.

Forward references: the `strtod`, `strtof`, and `strtold` functions (7.22.1.5), the `strtol`, `strtoll`, `strtoul`, and `strtoull` functions (7.22.1.7), conversion state (7.29.6), the `wcrtomb` function (7.29.6.3.3).

### 7.21.6.3 The `printf` function

**Synopsis**

```c
#include <stdio.h>
int printf(const char * restrict format, ...);
```

**Description**

The `printf` function is equivalent to `fprintf` with the argument `stdout` interposed before the arguments to `printf`.

**Returns**

The `printf` function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

### 7.21.6.4 The `scanf` function

**Synopsis**

```c
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

**Description**

The `scanf` function is equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`.
Returns
3 The scanf function returns the value of the macro EOF if an input failure occurs before the first conversion (if any) has completed. Otherwise, the scanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.5 The snprintf function
Synopsis
1 #include <stdio.h>
   int snprintf(char * restrict s, size_t n, const char * restrict format, ...);

Description
2 The snprintf function is equivalent to fprintf, except that the output is written into an array (specified by argument s) rather than to a stream. If n is zero, nothing is written, and s may be a null pointer. Otherwise, output characters beyond the n-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The snprintf function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

7.21.6.6 The sprintf function
Synopsis
1 #include <stdio.h>
   int sprintf(char * restrict s, const char * restrict format, ...);

Description
2 The sprintf function is equivalent to fprintf, except that the output is written into an array (specified by the argument s) rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The sprintf function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.7 The sscanf function
Synopsis
1 #include <stdio.h>
   int sscanf(const char * restrict s, const char * restrict format, ...);

Description
2 The sscanf function is equivalent to fscanf, except that input is obtained from a string (specified by the argument s) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the fscanf function. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The sscanf function returns the value of the macro EOF if an input failure occurs before the first conversion (if any) has completed. Otherwise, the sscanf function returns the number of input
items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.8 The `vfprintf` function

Synopsis

```c
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);
```

Description

The `vfprintf` function is equivalent to `fprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf` function does not invoke the `va_end` macro.\(^{308}\)

Returns

The `vfprintf` function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

EXAMPLE The following shows the use of the `vfprintf` function in a general error-reporting routine.

```c
#include <stdarg.h>
#include <stdio.h>

void error(char *function_name, char *format, ...)
{
    va_list args;
    va_start(args, format);
    // print out name of function causing error
    fprintf(stderr, "ERROR in %s: ", function_name);
    // print out remainder of message
    vfprintf(stderr, format, args);
    va_end(args);
}
```

7.21.6.9 The `vfscanf` function

Synopsis

```c
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);
```

Description

The `vfscanf` function is equivalent to `fscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfscanf` function does not invoke the `va_end` macro.\(^{308}\)

Returns

The `vfscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vfscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.10 The `vprintf` function

\(^{308}\)As the functions `vfprintf`, `vfscanf`, `vprintf`, `vscanf`, `vsnprintf`, `vsprintf`, and `vsscanf` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.
Synopsis
1
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char * restrict format, va_list arg);

Description
2
The vprintf function is equivalent to printf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vprintf function does not invoke the va_end macro.

Returns
3
The vprintf function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

7.21.6.11 The vscanf function
Synopsis
1
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char * restrict format, va_list arg);

Description
2
The vscanf function is equivalent to scanf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vscanf function does not invoke the va_end macro.

Returns
3
The vscanf function returns the value of the macro EOF if an input failure occurs before the first conversion (if any) has completed. Otherwise, the vscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.6.12 The vsnprintf function
Synopsis
1
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);

Description
2
The vsnprintf function is equivalent to snprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsnprintf function does not invoke the va_end macro. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3
The vsnprintf function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than n.

7.21.6.13 The vsprintf function
Synopsis
1
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char * restrict s, const char * restrict format, va_list arg);

Description
2 The vsprintf function is equivalent to sprintf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsprintf function does not invoke the va_end macro.\(^{308}\) If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The vsprintf function returns the number of characters written in the array, not counting the terminating null character, or a negative value if an encoding error occurred.

7.21.6.14 The vsscanf function
Synopsis
\#include <stdarg.h>
\#include <stdio.h>
int vsscanf(const char * restrict s, const char * restrict format, va_list arg);

Description
2 The vsscanf function is equivalent to sscanf, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vsscanf function does not invoke the va_end macro.\(^{308}\)

Returns
3 The vsscanf function returns the value of the macro EOF if an input failure occurs before the first conversion (if any) has completed. Otherwise, the vsscanf function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.21.7 Character input/output functions
7.21.7.1 The fgetc function
Synopsis
\#include <stdio.h>
int fgetc(FILE *stream);

Description
2 If the end-of-file indicator for the input stream pointed to by stream is not set and a next character is present, the fgetc function obtains that character as an unsigned char converted to an int and advances the associated file position indicator for the stream (if defined).

Returns
3 If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the fgetc function returns EOF. Otherwise, the fgetc function returns the next character from the input stream pointed to by stream. If a read error occurs, the error indicator for the stream is set and the fgetc function returns EOF.\(^{309}\)

7.21.7.2 The fgets function
Synopsis
\#include <stdio.h>
char *fgets(char * restrict s, int n, FILE * restrict stream);

\(^{309}\)An end-of-file and a read error can be distinguished by use of the feof and ferror functions.
The `fgets` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

The `fputc` function writes the character specified by `c` (converted to an `unsigned char`) to the output stream pointed to by `stream`, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The `fputc` function returns the character written. If a write error occurs, the error indicator for the stream is set and `fputc` returns `EOF`.

The `fputs` function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null character is not written.

The `fputs` function returns `EOF` if a write error occurs; otherwise it returns a nonnegative value.

The `getc` function is equivalent to `fgetc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects.

The `getc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getc` returns `EOF`. If a read error occurs, the error indicator for the stream is set and `getc` returns `EOF`.
7.21.7.6 The getchar function

Synopsis

```c
#include <stdio.h>
int getchar(void);
```

Description

The `getchar` function is equivalent to `getc` with the argument `stdin`.

Returns

The `getchar` function returns the next character from the input stream pointed to by `stdin`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getchar` returns `EOF`. If a read error occurs, the error indicator for the stream is set and `getchar` returns `EOF`.

7.21.7.7 The putc function

Synopsis

```c
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description

The `putc` function is equivalent to `fputc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so that argument should never be an expression with side effects.

Returns

The `putc` function returns the character written. If a write error occurs, the error indicator for the stream is set and `putc` returns `EOF`.

7.21.7.8 The putchar function

Synopsis

```c
#include <stdio.h>
int putchar(int c);
```

Description

The `putchar` function is equivalent to `putc` with the second argument `stdout`.

Returns

The `putchar` function returns the character written. If a write error occurs, the error indicator for the stream is set and `putchar` returns `EOF`. 
7.21.7.9 The puts function

Synopsis

```c
#include <stdio.h>
int puts(const char *s);
```

Description

The puts function writes the string pointed to by `s` to the stream pointed to by `stdout`, and appends a new-line character to the output. The terminating null character is not written.

Returns

The puts function returns EOF if a write error occurs; otherwise it returns a nonnegative value.

7.21.7.10 The ungetc function

Synopsis

```c
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description

The ungetc function pushes the character specified by `c` (converted to an unsigned char) back onto the input stream pointed to by `stream`. Pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file positioning function (`fseek`, `fsetpos`, or `rewind`) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the ungetc function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the ungetc function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back.\(^{310}\) For a text stream, the value of its file position indicator after a successful call to the ungetc function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the ungetc function; if its value was zero before a call, it is indeterminate after the call.\(^{311}\)

Returns

The ungetc function returns the character pushed back after conversion, or EOF if the operation fails.

Forward references: file positioning functions (7.21.9).

7.21.8 Direct input/output functions

7.21.8.1 The fread function

Synopsis

```c
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict stream);
```

\(^{310}\)Note that a file positioning function could further modify the file position indicator after discarding any pushed-back characters.

\(^{311}\)See “future library directions” (7.31.13).
Description
2 The `fread` function reads, into the array pointed to by `ptr`, up to `nmemb` elements whose size is
specified by `size`, from the stream pointed to by `stream`. For each object, `size` calls are made to
the `fgetc` function and the results stored, in the order read, in an array of `unsigned char` exactly
overlaying the object. The file position indicator for the stream (if defined) is advanced by the
number of characters successfully read. If an error occurs, the resulting value of the file position
indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

Returns
3 The `fread` function returns the number of elements successfully read, which may be less than `nmemb`
if a read error or end-of-file is encountered. If `size` or `nmemb` is zero, `fread` returns zero and the
contents of the array and the state of the stream remain unchanged.

7.21.8.2 The `fwrite` function

Synopsis
1
```
#include <stdio.h>
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb, FILE * restrict stream);
```

Description
2 The `fwrite` function writes, from the array pointed to by `ptr`, up to `nmemb` elements whose size is
specified by `size`, to the stream pointed to by `stream`. For each object, `size` calls are made to the
`fputc` function, taking the values (in order) from an array of `unsigned char` exactly overlaying the
object. The file position indicator for the stream (if defined) is advanced by the number of characters
successfully written. If an error occurs, the resulting value of the file position indicator for the stream
is indeterminate.

Returns
3 The `fwrite` function returns the number of elements successfully written, which will be less than
`nmemb` only if a write error is encountered. If `size` or `nmemb` is zero, `fwrite` returns zero and the
state of the stream remains unchanged.

7.21.9 File positioning functions

7.21.9.1 The `fgetpos` function

Synopsis
1
```
#include <stdio.h>
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

Description
2 The `fgetpos` function stores the current values of the parse state (if any) and file position indicator
for the stream pointed to by `stream` in the object pointed to by `pos`. The values stored contain
unspecified information usable by the `fsetpos` function for repositioning the stream to its position
at the time of the call to the `fgetpos` function.

Returns
3 If successful, the `fgetpos` function returns zero; on failure, the `fgetpos` function returns nonzero
and stores an implementation-defined positive value in `errno`.

Forward references: the `fsetpos` function (7.21.9.3).
7.21.9.2 The fseek function

Synopsis

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

2 The fseek function sets the file position indicator for the stream pointed to by stream. If a read or write error occurs, the error indicator for the stream is set and fseek fails.

3 For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding offset to the position specified by whence. The specified position is the beginning of the file if whence is SEEK_SET, the current value of the file position indicator if SEEK_CUR, or end-of-file if SEEK_END. A binary stream need not meaningfully support fseek calls with a whence value of SEEK_END.

4 For a text stream, either offset shall be zero, or offset shall be a value returned by an earlier successful call to the ftell function on a stream associated with the same file and whence shall be SEEK_SET.

5 After determining the new position, a successful call to the fseek function undoes any effects of the ungetc function on the stream, clears the end-of-file indicator for the stream, and then establishes the new position. After a successful fseek call, the next operation on an update stream may be either input or output.

Returns

6 The fseek function returns nonzero only for a request that cannot be satisfied.

Forward references: the ftell function (7.21.9.4).

7.21.9.3 The fsetpos function

Synopsis

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

2 The fsetpos function sets the mbstate_t object (if any) and file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which shall be a value obtained from an earlier successful call to the fgetpos function on a stream associated with the same file. If a read or write error occurs, the error indicator for the stream is set and fsetpos fails.

3 A successful call to the fsetpos function undoes any effects of the ungetc function on the stream, clears the end-of-file indicator for the stream, and then establishes the new parse state and position. After a successful fsetpos call, the next operation on an update stream may be either input or output.

Returns

4 If successful, the fsetpos function returns zero; on failure, the fsetpos function returns nonzero and stores an implementation-defined positive value in errno.

7.21.9.4 The ftell function

Synopsis

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

2 The ftell function obtains the current value of the file position indicator for the stream pointed to by stream. For a binary stream, the value is the number of characters from the beginning of the file.

§ 7.21.9.4  Library 297
For a text stream, its file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

**Returns**
3 If successful, the `ftell` function returns the current value of the file position indicator for the stream. On failure, the `ftell` function returns −1L and stores an implementation-defined positive value in `errno`.

### 7.21.9.5 The `rewind` function

**Synopsis**

```c
#include <stdio.h>
void rewind(FILE *stream);
```

**Description**
2 The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```c
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

**Returns**
3 The `rewind` function returns no value.

### 7.21.10 Error-handling functions

#### 7.21.10.1 The `clearerr` function

**Synopsis**

```c
#include <stdio.h>
void clearerr(FILE *stream);
```

**Description**
2 The `clearerr` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

**Returns**
3 The `clearerr` function returns no value.

#### 7.21.10.2 The `feof` function

**Synopsis**

```c
#include <stdio.h>
int feof(FILE *stream);
```

**Description**
2 The `feof` function tests the end-of-file indicator for the stream pointed to by `stream`.

**Returns**
3 The `feof` function returns nonzero if and only if the end-of-file indicator is set for `stream`. 
7.21.10.3 The `ferror` function

Synopsis

```
#include <stdio.h>
int ferror(FILE *stream);
```

Description

The `ferror` function tests the error indicator for the stream pointed to by `stream`.

Returns

The `ferror` function returns nonzero if and only if the error indicator is set for `stream`.

7.21.10.4 The `perror` function

Synopsis

```
#include <stdio.h>
void perror(const char *s);
```

Description

The `perror` function maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first (if `s` is not a null pointer and the character pointed to by `s` is not the null character), the string pointed to by `s` followed by a colon (:) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message strings are the same as those returned by the `strerror` function with argument `errno`.

Returns

The `perror` function returns no value.

Forward references: the `strerror` function (7.24.6.2).
7.22 General utilities <stdlib.h>

The header <stdlib.h> declares five types and several functions of general utility, and defines several macros.\footnote{See “future library directions” (7.31.14).}

The feature test macro \_\_STDC\_VERSION\_STDLIB\_H\_ expands to the token yyyymmL.

The types declared are \texttt{size\_t} and \texttt{wchar\_t} (both described in 7.19),

\begin{verbatim}
struct div_t {
  int quot;
  int rem;
};
struct ldiv_t {
  long quot;
  long rem;
};
struct lldiv_t {
  long long quot;
  long long rem;
};
\end{verbatim}

which is a structure type that is the type of the value returned by the \texttt{div} function,

\begin{verbatim}
struct ldiv_t {
  long quot;
  long rem;
};
\end{verbatim}

which is a structure type that is the type of the value returned by the \texttt{ldiv} function, and

\begin{verbatim}
struct lldiv_t {
  long long quot;
  long long rem;
};
\end{verbatim}

which is a structure type that is the type of the value returned by the \texttt{lldiv} function.

The macros defined are \texttt{NULL} (described in 7.19);

\begin{verbatim}
#define EXIT_FAILURE 0
#define EXIT_SUCCESS 1
\end{verbatim}

and

\begin{verbatim}
#define RAND_MAX size_t
#define MB_CUR_MAX size_t
\end{verbatim}

which expand to integer constant expressions that can be used as the argument to the \texttt{exit} function to return unsuccessful or successful termination status, respectively, to the host environment;

\begin{verbatim}
#define RAND_MAX size_t
\end{verbatim}

which expands to an integer constant expression that is the maximum value returned by the \texttt{rand} function; and

\begin{verbatim}
#define MB_CUR_MAX size_t
\end{verbatim}

which expands to a positive integer expression with type \texttt{size\_t} that is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category \texttt{LC\_CTYPE}), which is never greater than \texttt{MB\_LEN\_MAX}.

7.22.1 Numeric conversion functions

The functions \texttt{atof}, \texttt{atoi}, \texttt{atol}, and \texttt{atoll} need not affect the value of the integer expression \texttt{errno} on an error. If the value of the result cannot be represented, the behavior is undefined.

7.22.1.1 The \texttt{atof} function

Synopsis

\begin{verbatim}
#include <stdlib.h>

double atof(const char *nptr);
\end{verbatim}

Description

The \texttt{atof} function converts the initial portion of the string pointed to by \texttt{nptr} to \texttt{double} representation. Except for the behavior on error, it is equivalent to

\begin{verbatim}
strtod(nptr, (char **)NULL)
\end{verbatim}
Returns

3 The \texttt{atof} function returns the converted value.

Forward references: the \texttt{strtod}, \texttt{strtof}, and \texttt{strtold} functions (7.22.1.5–7.22.1.5).

7.22.1.2 The \texttt{atoi}, \texttt{atol}, and \texttt{atoll} functions

Synopsis

\begin{verbatim}
#include <stdlib.h>
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
\end{verbatim}

Description

2 The \texttt{atoi}, \texttt{atol}, and \texttt{atoll} functions convert the initial portion of the string pointed to by \texttt{nptr} to \texttt{int}, \texttt{long int}, and \texttt{long long int} representation, respectively. Except for the behavior on error, they are equivalent to

\begin{verbatim}
atoi: (int)strtol(nptr, (char **)NULL, 10) 
atol: strtol(nptr, (char **)NULL, 10) 
atoll: strtol(nptr, (char **)NULL, 10)
\end{verbatim}

Returns

3 The \texttt{atoi}, \texttt{atol}, and \texttt{atoll} functions return the converted value.

Forward references: the \texttt{strtol}, \texttt{strtoll}, \texttt{strtoul}, and \texttt{stroull} functions (7.22.1.7–7.22.1.7).

7.22.1.3 The \texttt{strfromd}, \texttt{strfromf}, and \texttt{strfroml} functions

Synopsis

\begin{verbatim}
#include <stdlib.h>
int strfromd(char *restrict s, size_t n, const char *restrict format, double fp);
int strfromf(char *restrict s, size_t n, const char *restrict format, float fp);
int strfroml(char *restrict s, size_t n, const char *restrict format, long double fp);
\end{verbatim}

Description

2 The \texttt{strfromd}, \texttt{strfromf}, and \texttt{strfroml} functions are equivalent to \texttt{snprintf(s, n, \texttt{format}, \texttt{fp})} (7.21.6.5), except that the format string shall only contain the character \texttt{%}, an optional precision that does not contain an asterisk \texttt{*}, and one of the conversion specifiers \texttt{a, A, e, E, f, F, g, or G}, which applies to the type (\texttt{double, float, or long double}) indicated by the function suffix (rather than by a length modifier).

Returns

The \texttt{strfromd}, \texttt{strfromf}, and \texttt{strfroml} functions return the number of characters that would have been written had \texttt{n} been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than \texttt{n}.

7.22.1.4 The \texttt{strfromdN} functions

Synopsis

\begin{verbatim}
#include <stdlib.h>
#define __STDC_IEC_60559_DFP__
int strfromd32(char *restrict s, size_t n, const char *restrict format, _Decimal32 fp);
int strfromd64(char *restrict s, size_t n, const char *restrict format, _Decimal64 fp);
int strfromd128(char *restrict s, size_t n, const char *restrict format, _Decimal128 fp);
#endif
\end{verbatim}

Description

\section*{§ 7.22.1.4 Library}
The `strtod`N functions are equivalent to `snprintf(s, n, format, fp)` (7.21.6.5), except the format string contains only the character %, an optional precision that does not contain an asterisk *, and one of the conversion specifiers a, A, e, E, f, F, g, or G, which applies to the type (_Decimal32, _Decimal64, or _Decimal128) indicated by the function suffix (rather than by a length modifier). Use of these functions with any other format string results in undefined behavior.

Returns

Synopsis replace. The `strtod`N functions return the number of characters that would have been written had n been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than n.

### 7.22.1.5 The `strtod`, `strtof`, and `strtold` functions

#### Synopsis

```c
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

#### Description

The `strtod`, `strtof`, and `strtold` functions convert the initial portion of the string pointed to by `nptr` to `double`, `float`, and `long double` representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2;
- a 0x or 0X, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary exponent part as defined in 6.4.4.2;
- INF or INFINITY, ignoring case
- NAN or NAN(n-char-sequence_opt), ignoring case in the NAN part, where:

```
  n-char-sequence:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point character is used in place of a period, and that if neither an exponent part nor a decimal-point character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence
is interpreted as negated. A character sequence INF or INFINITY is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A character sequence NAN or NaN(n-char-sequenceopt) is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequence is implementation-defined. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

5 If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

6 In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

7 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Recommended practice

8 If the subject sequence has the hexadecimal form, FLT_RADIX is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

9 If the subject sequence has the decimal form and at most \( M \) significant digits, where \( M \) is the maximum value of the \_DECIMAL_DIG macros (defined in \<float.h\>) \( \leq \) significant digits, the result should be correctly rounded. If the subject sequence \( D \) has the decimal form and more than \_DECIMAL_DIG \( M \) significant digits, consider the two bounding, adjacent decimal strings \( L \) and \( U \), both having \_DECIMAL_DIG \( M \) significant digits, such that the values of \( L \), \( D \), and \( U \) satisfy \( L \leq D \leq U \). The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding \( L \) and \( U \) according to the current rounding direction, with the extra stipulation that the error with respect to \( D \) should have a correct sign for the current rounding direction.

Returns

The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus HUGE_VAL, HUGE_VALF, or HUGE_VALL is returned (according to the return type and sign of the value), and the value of the macro ERANGE is stored in errno. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether errno acquires the value ERANGE is implementation-defined.

7.22.1.6 The strtodN functions

Synopsis

```c
#include <stdlib.h>
#endif
#define _STDC_IEC_60559_DFP_

_Decimal32 strtod32(const char * restrict nptr, char ** restrict endptr);
_Decimal64 strtod64(const char * restrict nptr, char ** restrict endptr);
_Decimal128 strtod128(const char * restrict nptr, char ** restrict endptr);
#endif
```

Description

2 The strtodN functions convert the initial portion of the string pointed to by nptr to decimal floating type representation. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters; a subject sequence resembling a floating constant or representing an infinity or NaN; and a final string of one or more unrecognized characters, including

---

313 It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods could yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.

314 An implementation can use the n-char sequence to determine extra information to be represented in the NaN’s significand.

315 \( M \) is sufficiently large that \( L \) and \( U \) will usually correctly round to the same internal floating value, but if not will correctly round to adjacent values.
the terminating null character of the input string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.4.4.2
- \text{INF} or \text{INFINITY}, ignoring case
- \text{NAN} or \text{NAN} (\text{d-char-sequence}_{opt}), ignoring case in the \text{NAN} part, where:
  
  \text{d-char-sequence}:
  
  digit
  
  nondigit
  
  d-char-sequence digit
  
  d-char-sequence nondigit

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

4 If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, including correct rounding and determination of the coefficient \(c\) and the quantum exponent \(q\), with the following exceptions:

- It is not a hexadecimal floating number.
- The decimal-point character is used in place of a period.
- If neither an exponent part nor a decimal-point character appears in a decimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated (before rounding) and the sign \(s\) is set to \(-1\), else \(s\) is set to \(1\). A character sequence \text{INF} or \text{INFINITY} is interpreted as an infinity. A character sequence \text{NAN} or \text{NAN} (\text{d-char-sequence}_{opt}), is interpreted as a quiet NaN; the meaning of the d-char sequence is implementation-defined.\(^{316}\) A pointer to the final string is stored in the object pointed to by \text{endptr}, provided that \text{endptr} is not a null pointer.

5 In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

6 If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of \(\text{nptr}\) is stored in the object pointed to by \text{endptr}, provided that \text{endptr} is not a null pointer.

Returns

7 The \text{strtod}\(N\) functions return the correctly rounded converted value, if any. If no conversion could be performed, the value of the triple \((1,0,0)\) is returned. If the correct value overflows, the value of the macro \text{ERANGE} is stored in \text{errno}. If the result underflows (7.12.1), whether \text{errno} acquires the value \text{ERANGE} is implementation-defined.

8 EXAMPLE Following are subject sequences of the decimal form and the resulting triples \((s, c, q)\) produced by \text{strtod64}.

Note that for \_Decimal64, the precision (maximum coefficient length) is \(16\) and the quantum exponent range is \(-398 \leq q \leq 369\).

\(^{316}\) An implementation may use the d-char sequence to determine extra information to be represented in the NaN's significand.
7.22.1.7 The <strong>strtol</strong>, <strong>strtoll</strong>, <strong>strtoul</strong>, and <strong>strtoull</strong> functions

Synopsis

```c
#include <stdlib.h>

long int strtol(const char *restrict nptr, char **restrict endptr, int base);
unsigned long int strtoul(const char *restrict nptr, char **restrict endptr, int base);
```

Description

The <strong>strtol</strong>, <strong>strtoll</strong>, <strong>strtoul</strong>, and <strong>strtoull</strong> functions convert the initial portion of the string pointed to by <code>nptr</code> to <code>long int</code>, <code>long long int</code>, <code>unsigned long int</code>, and <code>unsigned long long int</code> representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the function), a subject sequence resembling an integer represented in some radix determined by the value of <code>base</code>, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of <code>base</code> is zero, the expected form of the subject sequence is that of an integer constant as described in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix.
If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Returns

The `strtol`, `strtol`, `strtoul`, and `strtoull` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, or `ULLONG_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

### 7.22.2 Pseudo-random sequence generation functions

#### 7.22.2.1 The `rand` function

Synopsis

```c
#include <stdlib.h>
int rand(void);
```

Description

The `rand` function computes a sequence of pseudo-random integers in the range 0 to `RAND_MAX` inclusive.

The `rand` function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the `rand` function.

Recommended practice

There are no guarantees as to the quality of the random sequence produced and some implementations are known to produce sequences with distressingly non-random low-order bits. Applications with particular requirements should use a generator that is known to be sufficient for their needs.

Returns

The `rand` function returns a pseudo-random integer.

Environmental limits

The value of the `RAND_MAX` macro shall be at least 32767.

#### 7.22.2.2 The `srand` function
Synopsis

```c
#include <stdlib.h>
void srand(unsigned int seed);
```

Description

2 The `srand` function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If `rand` is called before any calls to `srand` have been made, the same sequence shall be generated as when `srand` is first called with a seed value of 1.

3 The `srand` function is not required to avoid data races with other calls to pseudo-random sequence generation functions. The implementation shall behave as if no library function calls the `srand` function.

Returns

4 The `srand` function returns no value.

EXAMPLE The following functions define a portable implementation of `rand` and `srand`.

```c
static unsigned long int next = 1;

int rand(void) // RAND_MAX assumed to be 32767
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
```

7.22.3 Memory management functions

The order and contiguity of storage allocated by successive calls to the `aligned_alloc`, `calloc`, `malloc`, and `realloc` functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and then size less than or equal to the size requested. It may then be used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

For purposes of determining the existence of a data race, memory allocation functions behave as though they accessed only memory locations accessible through their arguments and not other static duration storage. These functions may, however, visibly modify the storage that they allocate or deallocate. Calls to these functions that allocate or deallocate a particular region of memory shall occur in a single total order, and each such deallocation call shall synchronize with the next allocation (if any) in this order.

7.22.3.1 The `aligned_alloc` function

Synopsis

```c
#include <stdlib.h>
void *aligned_alloc(size_t alignment, size_t size);
```
Description
2 The **aligned_alloc** function allocates space for an object whose alignment is specified by `alignment`, whose size is specified by `size`, and whose value is indeterminate. If the value of `alignment` is not a valid alignment supported by the implementation the function shall fail by returning a null pointer.

Returns
3 The **aligned_alloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.2 The **calloc** function
Synopsis
1 
```c
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Description
2 The **calloc** function allocates space for an array of `nmemb` objects, each of whose size is `size`. The space is initialized to all bits zero.\(^\text{317}\)

Returns
3 The **calloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.3 The **free** function
Synopsis
1 
```c
#include <stdlib.h>
void free(void *ptr);
```

Description
2 The **free** function causes the space pointed to by `ptr` to be deallocated, that is, made available for further allocation. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns
3 The **free** function returns no value.

7.22.3.4 The **malloc** function
Synopsis
1 
```c
#include <stdlib.h>
void *malloc(size_t size);
```

Description
2 The **malloc** function allocates space for an object whose size is specified by `size` and whose value is indeterminate.

Returns
3 The **malloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.5 The **realloc** function
Synopsis
1 
```c
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

\(^{317}\)Note that this need not be the same as the representation of floating-point zero or a null pointer constant.
Description

2 The realloc function deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values.

3 If ptr is a null pointer, the realloc function behaves like the malloc function for the specified size. Otherwise, if ptr does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to the free or realloc function, the behavior is undefined. If size is nonzero and memory for the new object is not allocated, the old object is not deallocated. If size is zero and memory for the new object is not allocated, it is implementation-defined whether the old object is deallocated. If the old object is not deallocated, its value shall be unchanged.

Returns

4 The realloc function returns a pointer to the new object (which may have the same value as a pointer to the old object), or a null pointer if the new object has not been allocated.

7.22.4 Communication with the environment

7.22.4.1 The abort function

Synopsis

```c
#include <stdlib.h>
_Noreturn void abort(void);
```

Description

2 The abort function causes abnormal program termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status unsuccessful termination is returned to the host environment by means of the function call raise(SIGABRT).

Returns

3 The abort function does not return to its caller.

7.22.4.2 The atexit function

Synopsis

```c
#include <stdlib.h>
int atexit(void (*func)(void));
```

Description

2 The atexit function registers the function pointed to by func, to be called without arguments at normal program termination. It is unspecified whether a call to the atexit function that does not happen before the exit function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The atexit function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the at_quick_exit function (7.22.4.3), the exit function (7.22.4.4).

7.22.4.3 The at_quick_exit function

The atexit function registrations are distinct from the at_quick_exit registrations, so applications might need to call both registration functions with the same argument.
Synopsis
1
```c
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

Description
2 The `at_quick_exit` function registers the function pointed to by `func`, to be called without arguments should `quick_exit` be called.\(^{319}\) It is unspecified whether a call to the `at_quick_exit` function that does not happen before the `quick_exit` function is called will succeed.

Environmental limits
3 The implementation shall support the registration of at least 32 functions.

Returns
4 The `at_quick_exit` function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the `quick_exit` function (7.22.4.7).

7.22.4.4 The `exit` function

Synopsis
1
```c
#include <stdlib.h>
_Noreturn void exit(int status);
```

Description
2 The `exit` function causes normal program termination to occur. No functions registered by the `at_quick_exit` function are called. If a program calls the `exit` function more than once, or calls the `quick_exit` function in addition to the `exit` function, the behavior is undefined.

3 First, all functions registered by the `atexit` function are called, in the reverse order of their registration,\(^{320}\) except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the `longjmp` function is made that would terminate the call to the registered function, the behavior is undefined.

4 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed.

5 Finally, control is returned to the host environment. If the value of `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status `successful termination` is returned. If the value of `status` is `EXIT_FAILURE`, an implementation-defined form of the status `unsuccessful termination` is returned. Otherwise the status returned is implementation-defined.

Returns
6 The `exit` function cannot return to its caller.

7.22.4.5 The `_Exit` function

Synopsis
1
```c
#include <stdlib.h>
_Noreturn void _Exit(int status);
```

Description
2 The `_Exit` function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the `atexit` function, the `at_quick_exit` function, or signal handlers registered by the `signal` function are called. The status returned to the host

\(^{319}\)The `at_quick_exit` function registrations are distinct from the `atexit` registrations, so applications might need to call both registration functions with the same argument.

\(^{320}\)Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.
environment is determined in the same way as for the `exit` function (7.22.4.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

**Returns**

The `_Exit` function cannot return to its caller.

### 7.22.4.6 The `getenv` function

**Synopsis**

```c
#include <stdlib.h>
char *getenv(const char *name);
```

**Description**

The `getenv` function searches an environment list, provided by the host environment, for a string that matches the string pointed to by `name`. The set of environment names and the method for altering the environment list are implementation-defined. The `getenv` function need not avoid data races with other threads of execution that modify the environment list.\(^{321}\)

The implementation shall behave as if no library function calls the `getenv` function.

**Returns**

The `getenv` function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified `name` cannot be found, a null pointer is returned.

### 7.22.4.7 The `quick_exit` function

**Synopsis**

```c
#include <stdlib.h>
_Noreturn void quick_exit(int status);
```

**Description**

The `quick_exit` function causes normal program termination to occur. No functions registered by the `atexit` function or signal handlers registered by the `signal` function are called. If a program calls the `quick_exit` function more than once, or calls the `exit` function in addition to the `quick_exit` function, the behavior is undefined. If a signal is raised while the `quick_exit` function is executing, the behavior is undefined.

The `quick_exit` function first calls all functions registered by the `at_quick_exit` function, in the reverse order of their registration,\(^{322}\) except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the `longjmp` function is made that would terminate the call to the registered function, the behavior is undefined.

Then control is returned to the host environment by means of the function call `_Exit(status)`.\(^{323}\)

**Returns**

The `quick_exit` function cannot return to its caller.

### 7.22.4.8 The `system` function

**Synopsis**

```c
#include <stdlib.h>
int system(const char *string);
```

---

\(^{321}\) Many implementations provide non-standard functions that modify the environment list.

\(^{322}\) Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.
Description
2 If string is a null pointer, the system function determines whether the host environment has a command processor. If string is not a null pointer, the system function passes the string pointed to by string to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling system to behave in a non-conforming manner or to terminate.

Returns
3 If the argument is a null pointer, the system function returns nonzero only if a command processor is available. If the argument is not a null pointer, and the system function does return, it returns an implementation-defined value.

7.22.5 Searching and sorting utilities
1 These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as size_t nmemb specifies the length of the array for a function, nmemb can have the value zero on a call to that function; the comparison function is not called, a search finds no matching element, and sorting performs no rearrangement. Pointer arguments on such a call shall still have valid values, as described in 7.1.4.
2 The implementation shall ensure that the second argument of the comparison function (when called from bsearch), or both arguments (when called from qsort), are pointers to elements of the array. The first argument when called from bsearch shall equal key.
3 The comparison function shall not alter the contents of the array. The implementation may reorder elements of the array between calls to the comparison function, but shall not alter the contents of any individual element.
4 When the same objects (consisting of size bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for qsort they shall define a total ordering on the array, and for bsearch the same object shall always compare the same way with the key.
5 A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

7.22.5.1 The bsearch function

Synopsis
1

```c
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

Description
2 The bsearch function searches an array of nmemb objects, the initial element of which is pointed to by base, for an element that matches the object pointed to by key. The size of each element of the array is specified by size.
3 The comparison function pointed to by compar is called with two arguments that point to the key object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the key object, in

323) That is, if the value passed is \( p \), then the following expressions are always nonzero:

```c
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```
that order. \(^{324}\)

**Returns**

The `bsearch` function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

\(^{324}\)In practice, the entire array is sorted according to the comparison function.
7.22.5.2 The qsort function

Synopsis
1
```c
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Description
2 The qsort function sorts an array of nmemb objects, the initial element of which is pointed to by base. The size of each object is specified by size.
3 The contents of the array are sorted into ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.
4 If two elements compare as equal, their order in the resulting sorted array is unspecified.

Returns
5 The qsort function returns no value.

7.22.6 Integer arithmetic functions

7.22.6.1 The abs, labs, and llabs functions

Synopsis
1
```c
#include <stdlib.h>
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
```

Description
2 The abs, labs, and llabs functions compute the absolute value of an integer j. If the result cannot be represented, the behavior is undefined."325"

Returns
3 The abs, labs, and llabs, functions return the absolute value.

7.22.6.2 The div, ldiv, and lldiv functions

Synopsis
1
```c
#include <stdlib.h>
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
```

Description
2 The div, ldiv, and lldiv, functions compute numer/denom and numer%denom in a single operation.

Returns
3 The div, ldiv, and lldiv functions return a structure of type div_t, ldiv_t, and lldiv_t, respectively, comprising both the quotient and the remainder. The structures shall contain (in either order) the members quot (the quotient) and rem (the remainder), each of which has the same type as the arguments numer and denom. If either part of the result cannot be represented, the behavior is undefined.

325 The absolute value of the most negative number cannot be represented in two’s complement.
7.22.7 Multibyte/wide character conversion functions

The behavior of the multibyte character functions is affected by the \texttt{LC\_CTYPE} category of the current locale. For a state-dependent encoding, each function of the \texttt{mbtowc} and \texttt{wctomb} functions is placed into its initial conversion state at program startup and can be returned to that state by a call for which its character pointer argument, \texttt{s}, is a null pointer. Subsequent calls with \texttt{s} as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with \texttt{s} as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise.\footnote{If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.} Changing the \texttt{LC\_CTYPE} category causes the conversion state of these \texttt{mbtowc} and \texttt{wctomb} functions to be indeterminate.

7.22.7.1 The \texttt{mblen} function

Synopsis

```c
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

If \texttt{s} is not a null pointer, the \texttt{mblen} function determines the number of bytes contained in the multibyte character pointed to by \texttt{s}. Except that the conversion state of the \texttt{mbtowc} function is not affected, it is equivalent to

\[
\text{mbtowc}((\texttt{wchar\_t} \ast)0, (\texttt{const char} \ast)0, 0);
\text{mbtowc}((\texttt{wchar\_t} \ast)0, \texttt{s}, \texttt{n});
\]

The implementation shall behave as if no library function calls the.

Returns

If \texttt{s} is a null pointer, the \texttt{mblen} function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If \texttt{s} is not a null pointer, the \texttt{mblen} function either returns 0 (if \texttt{s} points to the null character), or returns the number of bytes that are contained in the multibyte character (if the next \texttt{n} or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

Forward references: the \texttt{mbtowc} function (7.22.7.2).

7.22.7.2 The \texttt{mbtowc} function

Synopsis

```c
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

Description

If \texttt{s} is not a null pointer, the \texttt{mbtowc} function inspects at most \texttt{n} bytes beginning with the byte pointed to by \texttt{s} to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if \texttt{pwc} is not a null pointer, stores that value in the object pointed to by \texttt{pwc}. If the corresponding wide character is the null wide character, the function is left in the initial conversion state.

The implementation shall behave as if no library function calls the \texttt{mbtowc} function.

Returns

If \texttt{s} is a null pointer, the \texttt{mbtowc} function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If \texttt{s} is not a null pointer, the \texttt{mbtowc} function either returns 0 (if \texttt{s} points to the null character), or returns the number of bytes
that are contained in the converted multibyte character (if the next \(n\) or fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than \(n\) or the value of the `MB_CUR_MAX` macro.

### 7.22.7.3 The wctomb function

**Synopsis**

```c
#include <stdlib.h>
int wctomb(char *s, wchar_t wc);
```

**Description**

The `wctomb` function determines the number of bytes needed to represent the multibyte character corresponding to the wide character given by `wc` (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by `s` (if `s` is not a null pointer). At most `MB_CUR_MAX` characters are stored. If `wc` is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.

The implementation shall behave as if no library function calls the `wctomb` function.

**Returns**

If `s` is a null pointer, the `wctomb` function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If `s` is not a null pointer, the `wctomb` function returns -1 if the value of `wc` does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character corresponding to the value of `wc`.

In no case will the value returned be greater than the value of the `MB_CUR_MAX` macro.

### 7.22.8 Multibyte/wide string conversion functions

The behavior of the multibyte string functions is affected by the `LC_CTYPE` category of the current locale.

#### 7.22.8.1 The mbstowcs function

**Synopsis**

```c
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

**Description**

The `mbstowcs` function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by `s` into a sequence of corresponding wide characters and stores not more than \(n\) wide characters into the array pointed to by `pwcs`. No multibyte characters that follow a null character (which is converted into a null wide character) will be examined or converted. Each multibyte character is converted as if by a call to the `mbtowc` function, except that the conversion state of the `mbtowc` function is not affected.

No more than \(n\) elements will be modified in the array pointed to by `pwcs`. If copying takes place between objects that overlap, the behavior is undefined.

**Returns**

If an invalid multibyte character is encountered, the `mbstowcs` function returns `(size_t)(-1)`. Otherwise, the `mbstowcs` function returns the number of array elements modified, not including a terminating null wide character, if any.\(^{327}\)

\(^{327}\)The array will not be null-terminated if the value returned is \(n\).
7.22.8.2 The `wcstombs` function

Synopsis

```c
#include <stdlib.h>
size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

Description

The `wcstombs` function converts a sequence of wide characters from the array pointed to by `pwcs` into a sequence of corresponding multibyte characters that begins in the initial shift state, and stores these multibyte characters into the array pointed to by `s`, stopping if a multibyte character would exceed the limit of `n` total bytes or if a null character is stored. Each wide character is converted as if by a call to the `wctomb` function, except that the conversion state of the `wctomb` function is not affected.

No more than `n` bytes will be modified in the array pointed to by `s`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

If a wide character is encountered that does not correspond to a valid multibyte character, the `wcstombs` function returns `(size_t)(-1)`. Otherwise, the `wcstombs` function returns the number of bytes modified, not including a terminating null character, if any.\(^{327}\)
7.23 __Noreturn <stdnoreturn.h>

The header <stdnoreturn.h> defines the macro

```c
noreturn
```

which expands to `__Noreturn`. 
7.24  String handling <string.h>

7.24.1  String function conventions

1  The header <string.h> declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.\(^{328}\) The type is \texttt{size_t} and the macro is \texttt{NULL} (both described in 7.19). Various methods are used for determining the lengths of the arrays, but in all cases a \texttt{char *} or \texttt{void *} argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2  Where an argument declared as \texttt{size_t n} specifies the length of the array for a function, \(n\) can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a character finds no occurrence, a function that compares two character sequences returns zero, and a function that copies characters copies zero characters.

3  For all functions in this subclause, each character shall be interpreted as if it had the type \texttt{unsigned char} (and therefore every possible object representation is valid and has a different value).

7.24.2  Copying functions

7.24.2.1  The \texttt{memcpy} function

Synopsis

```
#include <string.h>
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

Description

2  The \texttt{memcpy} function copies \(n\) characters from the object pointed to by \(s2\) into the object pointed to by \(s1\). If copying takes place between objects that overlap, the behavior is undefined.

Returns

3  The \texttt{memcpy} function returns the value of \(s1\).

7.24.2.2  The \texttt{memccpy} function

Synopsis

```
#include <string.h>
void *memccpy(void * restrict s1, const void * restrict s2, int c, size_t n);
```

Description

2  The \texttt{memccpy} function copies characters from the object pointed to by \(s2\) into the object pointed to by \(s1\), stopping after the first occurrence of character \(c\) (converted to an \texttt{unsigned char}) is copied, or after \(n\) characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

Returns

3  The \texttt{memccpy} function returns a pointer to the character after the copy of \(c\) in \(s1\), or a null pointer if \(c\) was not found in the first \(n\) characters of \(s2\).

7.24.2.3  The \texttt{memmove} function

Synopsis

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

\(^{328}\)See “future library directions” (7.31.15).
Description
2 The `memmove` function copies $n$ characters from the object pointed to by $s2$ into the object pointed to by $s1$. Copying takes place as if the $n$ characters from the object pointed to by $s2$ are first copied into a temporary array of $n$ characters that does not overlap the objects pointed to by $s1$ and $s2$, and then the $n$ characters from the temporary array are copied into the object pointed to by $s1$.

Returns
3 The `memmove` function returns the value of $s1$.

7.24.2.4 The `strcpy` function
Synopsis
1

```
#include <string.h>
char *strcpy(char * restrict s1, const char * restrict s2);
```

Description
2 The `strcpy` function copies the string pointed to by $s2$ (including the terminating null character) into the array pointed to by $s1$. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The `strcpy` function returns the value of $s1$.

7.24.2.5 The `strncpy` function
Synopsis
1

```
#include <string.h>
char *strncpy(char * restrict s1, const char * restrict s2, size_t n);
```

Description
2 The `strncpy` function copies not more than $n$ characters (characters that follow a null character are not copied) from the array pointed to by $s2$ to the array pointed to by $s1$. If copying takes place between objects that overlap, the behavior is undefined.

3 If the array pointed to by $s2$ is a string that is shorter than $n$ characters, null characters are appended to the copy in the array pointed to by $s1$, until $n$ characters in all have been written.

Returns
4 The `strncpy` function returns the value of $s1$.

7.24.3 Concatenation functions
7.24.3.1 The `strcat` function
Synopsis
1

```
#include <string.h>
char *strcat(char * restrict s1, const char * restrict s2);
```

Description
2 The `strcat` function appends a copy of the string pointed to by $s2$ (including the terminating null character) to the end of the string pointed to by $s1$. The initial character of $s2$ overwrites the null character at the end of $s1$. If copying takes place between objects that overlap, the behavior is undefined.

Returns
3 The `strcat` function returns the value of $s1$.

---

329) Thus, if there is no null character in the first $n$ characters of the array pointed to by $s2$, the result will not be null-terminated.
7.24.3.2 The `strncat` function

Synopsis

```c
#include <string.h>
char *strncat(char * restrict s1, const char * restrict s2, size_t n);
```

Description

The `strncat` function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result.\(^{330}\) If copying takes place between objects that overlap, the behavior is undefined.

Returns

The `strncat` function returns the value of `s1`.

Forward references: the `strlen` function (7.24.6.3).

7.24.4 Comparison functions

The sign of a nonzero value returned by the comparison functions `memcmp`, `strcmp`, and `strncmp` is determined by the sign of the difference between the values of the first pair of characters (both interpreted as `unsigned char`) that differ in the objects being compared.

7.24.4.1 The `memcmp` function

Synopsis

```c
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

The `memcmp` function compares the first `n` characters of the object pointed to by `s1` to the first `n` characters of the object pointed to by `s2`\(^ {331}\).

Returns

The `memcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

7.24.4.2 The `strcmp` function

Synopsis

```c
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description

The `strcmp` function compares the string pointed to by `s1` to the string pointed to by `s2`.

Returns

The `strcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

7.24.4.3 The `strcoll` function

Synopsis

```c
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

\(^{330}\)Thus, the maximum number of characters that can end up in the array pointed to by `s1` is `strlen(s1)+n+1`.

\(^{331}\)The contents of “holes” used as padding for purposes of alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions can also cause problems in comparison.
Description

2 The **strcoll** function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

Returns

3 The **strcoll** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` when both are interpreted as appropriate to the current locale.

### 7.24.4.4 The **strncmp** function

**Synopsis**

```c
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

**Description**

2 The **strncmp** function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

**Returns**

3 The **strncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

### 7.24.4.5 The **strxfrm** function

**Synopsis**

```c
#include <string.h>
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

**Description**

2 The **strxfrm** function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

**Returns**

3 The **strxfrm** function returns the length of the transformed string (not including the terminating null character). If the value returned is `n` or more, the contents of the array pointed to by `s1` are indeterminate.

4 **EXAMPLE** The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by `s`.

```c
1 + strxfrm(NULL, s, 0)
```

### 7.24.5 Search functions

#### 7.24.5.1 The **memchr** function

**Synopsis**

```c
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```
Description

The **memchr** function locates the first occurrence of \( c \) (converted to an **unsigned char**) in the initial \( n \) characters (each interpreted as **unsigned char**) of the object pointed to by \( s \). The implementation shall behave as if it reads the characters sequentially and stops as soon as a matching character is found.

Returns

The **memchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

7.24.5.2 The **strchr** function

Synopsis

```c
#include <string.h>
char *strchr(const char *s, int c);
```

Description

The **strchr** function locates the first occurrence of \( c \) (converted to a **char**) in the string pointed to by \( s \). The terminating null character is considered to be part of the string.

Returns

The **strchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

7.24.5.3 The **strcspn** function

Synopsis

```c
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description

The **strcspn** function computes the length of the maximum initial segment of the string pointed to by \( s1 \) which consists entirely of characters **not** from the string pointed to by \( s2 \).

Returns

The **strcspn** function returns the length of the segment.

7.24.5.4 The **strpbrk** function

Synopsis

```c
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

The **strpbrk** function locates the first occurrence in the string pointed to by \( s1 \) of any character from the string pointed to by \( s2 \).

Returns

The **strpbrk** function returns a pointer to the character, or a null pointer if no character from \( s2 \) occurs in \( s1 \).

7.24.5.5 The **strrchr** function

Synopsis

```c
#include <string.h>
char *strrchr(const char *s, int c);
```
Description
2 The **strchr** function locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Returns
3 The **strchr** function returns a pointer to the character, or a null pointer if **c** does not occur in the string.

### 7.24.5.6 The **strspn** function

**Synopsis**
1
```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description
2 The **strspn** function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Returns
3 The **strspn** function returns the length of the segment.

### 7.24.5.7 The **strstr** function

**Synopsis**
1
```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description
2 The **strstr** function locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**.

Returns
3 The **strstr** function returns a pointer to the located string, or a null pointer if the string is not found. If **s2** points to a string with zero length, the function returns **s1**.

### 7.24.5.8 The **strtok** function

**Synopsis**
1
```
#include <string.h>
char *strtok(char * restrict s1, const char * restrict s2);
```

Description
2 A sequence of calls to the **strtok** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has a non-null first argument; subsequent calls in the sequence have a null first argument. The separator string pointed to by **s2** may be different from call to call.
3 The first call in the sequence searches the string pointed to by **s1** for the first character that is not contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.
4 The **strtok** function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.
Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The `strtok` function is not required to avoid data races with other calls to the `strtok` function. The implementation shall behave as if no library function calls the `strtok` function.

**Returns**

The `strtok` function returns a pointer to the first character of a token, or a null pointer if there is no token.

**EXAMPLE**

```c
#include <string.h>
static char str[] = "?a???b,,,#c";
char *t;

    t = strtok(str, "?");   // t points to the token "a"
    t = strtok(NULL, ",");   // t points to the token "??b"
    t = strtok(NULL, ",#");  // t points to the token "c"
    t = strtok(NULL, "?");  // t is a null pointer
```

Forward references: The `strtok_s` function (K.3.7.3.1).

### 7.24.6 Miscellaneous functions

#### 7.24.6.1 The `memset` function

**Synopsis**

```c
#include <string.h>
void *memset(void *s, int c, size_t n);
```

**Description**

The `memset` function copies the value of `c` (converted to an `unsigned char`) into each of the first `n` characters of the object pointed to by `s`.

**Returns**

The `memset` function returns the value of `s`.

#### 7.24.6.2 The `strerror` function

**Synopsis**

```c
#include <string.h>
char *strerror(int errnum);
```

**Description**

The `strerror` function maps the number in `errnum` to a message string. Typically, the values for `errnum` come from `errno`, but `strerror` shall map any value of type `int` to a message.

The `strerror` function is not required to avoid data races with other calls to the `strerror` function. The implementation shall behave as if no library function calls the `strerror` function.

**Returns**

The `strerror` function returns a pointer to the string, the contents of which are locale-specific. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

Forward references: The `strerror_s` function (K.3.7.4.2).

---

332) The `strtok_s` function can be used instead to avoid data races.
333) The `strerror_s` function can be used instead to avoid data races.
7.24.6.3 The `strlen` function

Synopsis

```c
#include <string.h>
size_t strlen(const char *s);
```

Description

The `strlen` function computes the length of the string pointed to by `s`.

Returns

The `strlen` function returns the number of characters that precede the terminating null character.

7.24.6.4 The `strdup` function

Synopsis

```c
#include <string.h>
char *strdup(const char *s);
```

Description

The `strdup` function creates a copy of the string pointed to by `s` in a space allocated as if by a call to `malloc`.

Returns

The `strdup` function returns a pointer to the first character of the duplicate string. The returned pointer can be passed to `free`. If no space can be allocated the `strdup` function returns a null pointer.

7.24.6.5 The `strndup` function

Synopsis

```c
#include <string.h>
char *strndup(const char *s, size_t size);
```

Description

The `strndup` function creates a string initialized with no more than `size` initial characters of the array pointed to by `s` and up to the first null character, whichever comes first, in a space allocated as if by a call to `malloc`. If the array pointed to by `s` does not contain a null within the first `size` characters, a null is appended to the copy of the array.

Returns

The `strndup` function returns a pointer to the first character of the created string. The returned pointer can be passed to `free`. If no space can be allocated the `strndup` function returns a null pointer.
7.25 Type-generic math <tgmath.h>

The header <tgmath.h> includes the headers <math.h> and <complex.h> and defines several type-generic macros.

The feature test macro __STDC_VERSION_TGMATH__ expands to the token yyyyMMdd.

This clause specifies a many-to-one correspondence of functions in <math.h> and <complex.h> with type-generic macros. Use of a type-generic macro invokes a corresponding function whose type is determined by the types of the arguments for particular parameters called the generic parameters.

Of the <math.h> and <complex.h> functions without an f(float) or l(long double) suffix, several have one or more parameters whose corresponding real type is double. For each such function, except modf, the functions that round result to narrower type (7.12.14) (which are covered below) and modf, there is a corresponding -type-generic macro. The parameters whose corresponding real type is double in the function synopsis are - Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters -generic parameters.

Some of the <math.h> functions for decimal floating types have no unsuffixed counterpart. Of these functions with a d64 suffix, some have one or more parameters whose type is _Decimal64. For each such function, except decodedecd64, encodedecd64, decodebind64, and encodebind64, there is a corresponding type-generic macro. The parameters whose real type is _Decimal64 in the function synopsis are generic parameters.

If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type, the behavior is undefined.

Use of the Except for the macros for functions that round result to a narrower type (7.12.14), use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the types of the arguments for the generic parameters as follows:

- First, Arguments of integer type are regarded as having type _Decimal64 if any argument for generic parameters has type long double, the type determined is long double, has decimal floating type, and as having type double otherwise.
- Otherwise, if any argument for generic parameters has type double or is of integer type, if the function has exactly one generic parameter, the type determined is the corresponding real type of the argument for the generic parameter.
- If the function has exactly two generic parameters, the type determined is double—the corresponding real type determined by the usual arithmetic conversions (6.3.1.8) applied to the arguments for the generic parameters.
- Otherwise, if the function has more than two generic parameters, the type determined is float—the corresponding real type determined by repeatedly applying the usual arithmetic conversions, first to the first two arguments for generic parameters, then to that result type and the next argument for a generic parameter, and so forth until the usual arithmetic conversions have been applied to the last argument for a generic parameter.

If neither <math.h> and <complex.h> define a function whose generic parameters have the determined corresponding real type, the behavior is undefined.

For each unsuffixed function in <math.h> for which there is a function in <complex.h> with the same name except for a <c prefix, the corresponding type-generic macro (for both functions) has

---

334 Like other function-like macros in standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

335 If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.
the same name as the function in `<math.h>`. The corresponding type-generic macro for `fabs` and `cabs` is `fabs` and `cabs`.

<table>
<thead>
<tr>
<th><code>&lt;math.h&gt;</code> function</th>
<th><code>&lt;complex.h&gt;</code> function</th>
<th>Type-generic macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>cacos</td>
<td>acos-acos</td>
</tr>
<tr>
<td>asin</td>
<td>casin</td>
<td>asin-asin</td>
</tr>
<tr>
<td>atan</td>
<td>catan</td>
<td>atan-atan</td>
</tr>
<tr>
<td>acosh</td>
<td>cacosh</td>
<td>acosh-acosh</td>
</tr>
<tr>
<td>asinh</td>
<td>casinh</td>
<td>asinh-asinh</td>
</tr>
<tr>
<td>atanh</td>
<td>catanh</td>
<td>atanh-atanh</td>
</tr>
<tr>
<td>cos</td>
<td>ccos</td>
<td>cos-cos</td>
</tr>
<tr>
<td>sin</td>
<td>csin</td>
<td>sin-sin</td>
</tr>
<tr>
<td>tan</td>
<td>ctan</td>
<td>tan-tan</td>
</tr>
<tr>
<td>cosh</td>
<td>ccosh</td>
<td>cosh-cosh</td>
</tr>
<tr>
<td>sinh</td>
<td>csinh</td>
<td>sinh-sinh</td>
</tr>
<tr>
<td>tanh</td>
<td>ctanh</td>
<td>tanh-tanh</td>
</tr>
<tr>
<td>exp</td>
<td>cexp</td>
<td>exp-exp</td>
</tr>
<tr>
<td>log</td>
<td>clog</td>
<td>log-log</td>
</tr>
<tr>
<td>pow</td>
<td>cpow</td>
<td>pow-pow</td>
</tr>
<tr>
<td>sqrt</td>
<td>csqrt</td>
<td>sqrt-sqrt</td>
</tr>
<tr>
<td>fabs</td>
<td>cabs</td>
<td>fabs-fabs</td>
</tr>
</tbody>
</table>

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

9 For each unsuffixed function in `<math.h>` without a `c-` prefixed counterpart in `<complex.h>` (except functions that round result to narrower type, `modf`, and `canonicalize`), the corresponding type-generic macro has the same name as the function. These type-generic macros are: `atan2`, `cbrt`, `ceil`, `copysign`, `erf`, `erfc`, `exp2`, `expm1`, `fdim`, `floor`, `fma`, `fmax`, `fmin`, `fmod`, `frexp`, `hypot`, `ilogb`, `ldexp`, `lgamma`, `llrint`, `llround`, `log10`, `log1p`, `log2`, `logb`, `lrint`, `lround`, `nearbyint`, `nextafter`, `nexttoward`.
For each unsuffixed function in `<complex.h>` that is not a `<math.h>`-prefixed counterpart to a function in `<math.h>`, the corresponding type-generic macro has the same name as the function. These type-generic macros are: `carg`.

```
carg  cimag  conj  cproj  creal
```

Use of the macro with any argument of standard floating or complex type invokes a complex function, `cimag`.

Use of the macro with an argument of decimal floating type is undefined.

The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with `f` or `d` prefix are:

```
fadd  fsub  fmul  fdiv  ffma  fsqrt
```

and the macros with `d32` or `d64` prefix are:

```
d32add  d32sub  d32mul  d32div  d32fma  d32sqrt
```

All arguments shall be real. `cproj`.

If the macro prefix is `f` or `d`, use of an argument of decimal floating type is undefined. If the macro prefix is `d32` or `d64`, use of an argument of standard floating type is undefined. The function invoked is determined as follows:

- If any argument has type `_Decimal128`, or if the macro prefix is `d64`, the function invoked has the name of the macro, with a `d128` suffix.
— Otherwise, if the macro prefix is `d32`, the function invoked has the name of the macro, with a `d64` suffix.

— Otherwise, if any argument has type `long double`, or if the macro prefix is `d`, the function invoked has the name of the macro, with an `l` suffix.

— Otherwise, the function invoked has the name of the macro (with no suffix).

ISO/IEC 9899:202x (E) working draft — November 18, 2019 C17..C2x N2455

12 `real` For each `d64-suffixed_function` in `<math.h>`, except `decodedecd64`, `encodedecd64`, `decodebind64`, and `encodebind64`, that does not have an unsuffixed counterpart, the corresponding type-generic macro has the name of the function, but without the suffix. These type-generic macros are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantized</td>
<td>quantize</td>
</tr>
<tr>
<td>samequantumed</td>
<td>samequantum</td>
</tr>
<tr>
<td>quantumed</td>
<td>quantum</td>
</tr>
<tr>
<td>llquantexpd</td>
<td>llquantexp</td>
</tr>
</tbody>
</table>

Use of the macro with any real or complex argument invokes a complex function—an argument of standard floating or complex type or with only integer type arguments is undefined.

13 A type-generic macro corresponding to a function indicated in the table in 7.6.2 is affected by constant rounding modes (7.6.4).

14 NOTE The type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows:

```c
#define cbrt(X)     \  
    _Generic((X), \  
        long double: _Roundwise_cbrtl, \  
        default: _Roundwise_cbrt, \  
        float: _Roundwise_cbrtf \  
    )(X)
```

where `_Roundwise_cbrtl`, `_Roundwise_cbrt`, and `_Roundwise_cbrtf` are pointers to functions that are equivalent to `cbrtl`, `cbrt`, and `cbrtf`, respectively, but that are guaranteed to be affected by constant rounding modes (7.6.2).
EXAMPLE With the declarations

```c
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
#endif
#define __STDC_IEC_60559_DFP__
#define _Decimal128 dc;
#define _Decimal64 d64;
#define _Decimal32 d32;
#endif
#include <tgmath.h>
```

functions invoked by use of type-generic macros are shown in the following table:

<table>
<thead>
<tr>
<th>macro use</th>
<th>invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp(n)</td>
<td>exp(n) exp(n) exp(n), the function</td>
</tr>
<tr>
<td>acosf(f)</td>
<td>acos(f) acosf(f)</td>
</tr>
<tr>
<td>sinf(d) sin(d)</td>
<td>sin(d) sinf(d)</td>
</tr>
<tr>
<td>atanf(ld atanf(ld)</td>
<td>atanf(ld atanf(ld)</td>
</tr>
<tr>
<td>logf(fc)</td>
<td>logf(fc) clogf(fc)</td>
</tr>
<tr>
<td>sqrt(dc sqrt(dc)</td>
<td>sqrt(dc) csqrt(dc)</td>
</tr>
<tr>
<td>pow(dc, f) pow(ldc, f)</td>
<td>cpow(ldc, f) cpow(ldc, f)</td>
</tr>
<tr>
<td>remainder(n, n)</td>
<td>remainder(n, n)</td>
</tr>
<tr>
<td>nextafter(d, f)</td>
<td>nextafter(d, f)</td>
</tr>
<tr>
<td>nexttoward(d64, ld)</td>
<td>nexttoward(d6, ld)</td>
</tr>
<tr>
<td>copysign(n, ld)</td>
<td>copysign(n, ld)</td>
</tr>
<tr>
<td>ceilf(fc)</td>
<td>ceil(fc)</td>
</tr>
<tr>
<td>sqrtf(dc)</td>
<td>sqrtf(dc)</td>
</tr>
<tr>
<td>fmax(ldc, ld) fmax(ldc, ld)</td>
<td>fmax(ldc, ld)</td>
</tr>
<tr>
<td>cprojf(f)</td>
<td>cproj(f) cprojf(f)</td>
</tr>
<tr>
<td>crealf(d)</td>
<td>creal(d)</td>
</tr>
<tr>
<td>cimagf(ld) cimagf(ld)</td>
<td>cimagf(ld)</td>
</tr>
<tr>
<td>fabsf(fc) fabsf(fc)</td>
<td>fabsf(fc)</td>
</tr>
<tr>
<td>cargf(fc)</td>
<td>carg(fc)</td>
</tr>
<tr>
<td>cprojf(ldc) cproj(ldc)</td>
<td>cprojf(ldc)</td>
</tr>
<tr>
<td>fdiv(d, f)</td>
<td>fdiv(d, f)</td>
</tr>
<tr>
<td>dfmaf(f, d, ld)</td>
<td>dfmaf(f, d, ld)</td>
</tr>
<tr>
<td>dadd(f, f)</td>
<td>dadd(f, f)</td>
</tr>
<tr>
<td>dsqrt(dc)</td>
<td>dsqrt(dc)</td>
</tr>
<tr>
<td>exp(d64)</td>
<td>expd64(d64)</td>
</tr>
<tr>
<td>sqrt(d32)</td>
<td>sqrt32(d32)</td>
</tr>
<tr>
<td>fmax(d64, d128)</td>
<td>fmaxd128(d64, d128)</td>
</tr>
<tr>
<td>powf(d32, n)</td>
<td>powd32(d32, n)</td>
</tr>
<tr>
<td>remainder(d64, d)</td>
<td>remainder(d64, d)</td>
</tr>
<tr>
<td>creal(d64)</td>
<td>creal(d64)</td>
</tr>
<tr>
<td>remquof(d32, d32, &amp;n)</td>
<td>remquof(d32, d32, &amp;n)</td>
</tr>
<tr>
<td>llquantexp(d)</td>
<td>llquantexp(d)</td>
</tr>
<tr>
<td>samequantum(n, n)</td>
<td>samequantum(n, n)</td>
</tr>
<tr>
<td>d32sub(d32, d128)</td>
<td>d32sub(d32, d128)</td>
</tr>
<tr>
<td>d32div(d64, n)</td>
<td>d32div(d64, n)</td>
</tr>
<tr>
<td>d64add(d32, d32)</td>
<td>d64add(d32, d32)</td>
</tr>
<tr>
<td>d64sqrt(d)</td>
<td>d64sqrt(d)</td>
</tr>
<tr>
<td>dadd(n, d64)</td>
<td>dadd(n, d64)</td>
</tr>
</tbody>
</table>
7.26 Threads <threads.h>

7.26.1 Introduction

The header <threads.h> includes the header <time.h>, defines macros, and declares types, enumeration constants, and functions that support multiple threads of execution.\(^{336}\)

Implementations that define the macro \texttt{__STDC_NO_THREADS__} need not provide this header nor support any of its facilities.

The macros are

\begin{verbatim}
thread_local
\end{verbatim}

which expands to the keyword \texttt{Thread_local};

\begin{verbatim}
ONCE_FLAG_INIT
\end{verbatim}

which expands to a value that can be used to initialize an object of type \texttt{once_flag}; and

\begin{verbatim}
TSS_DTOR_ITERATIONS
\end{verbatim}

which expands to an integer constant expression representing the maximum number of times that destructors will be called when a thread terminates.

The types are

\begin{verbatim}
cnd_t
\end{verbatim}

which is a complete object type that holds an identifier for a condition variable;

\begin{verbatim}
thrd_t
\end{verbatim}

which is a complete object type that holds an identifier for a thread;

\begin{verbatim}
tss_t
\end{verbatim}

which is a complete object type that holds an identifier for a thread-specific storage pointer;

\begin{verbatim}
mtx_t
\end{verbatim}

which is a complete object type that holds an identifier for a mutex;

\begin{verbatim}
tss_dtor_t
\end{verbatim}

which is the function pointer type \texttt{void (*) (void*)}, used for a destructor for a thread-specific storage pointer;

\begin{verbatim}
thrd_start_t
\end{verbatim}

which is the function pointer type \texttt{int (*) (void*)} that is passed to \texttt{thrd_create} to create a new thread; and

\begin{verbatim}
once_flag
\end{verbatim}

which is a complete object type that holds a flag for use by \texttt{call_once}.

The enumeration constants are

\begin{verbatim}
mtx_plain
\end{verbatim}

\(^{336}\)See “future library directions” (7.31.17).
which is passed to `mntx_init` to create a mutex object that does not support timeout;

```
mtx_recursive
```

which is passed to `mntx_init` to create a mutex object that supports recursive locking;

```
mtx_timed
```

which is passed to `mntx_init` to create a mutex object that supports timeout;

```
thrd_timedout
```

which is returned by a timed wait function to indicate that the time specified in the call was reached without acquiring the requested resource;

```
thrd_success
```

which is returned by a function to indicate that the requested operation succeeded;

```
thrd_busy
```

which is returned by a function to indicate that the requested operation failed because a resource requested by a test and return function is already in use;

```
thrd_error
```

which is returned by a function to indicate that the requested operation failed; and

```
thrd_nomem
```

which is returned by a function to indicate that the requested operation failed because it was unable to allocate memory.

**Forward references:** date and time (7.27).

### 7.26.2 Initialization functions

#### 7.26.2.1 The `call_once` function

**Synopsis**

```c
#include <threads.h>

void call_once(once_flag *flag, void (*func)(void));
```

**Description**

The `call_once` function uses the `once_flag` pointed to by `flag` to ensure that `func` is called exactly once, the first time the `call_once` function is called with that value of `flag`. Completion of an effective call to the `call_once` function synchronizes with all subsequent calls to the `call_once` function with the same value of `flag`.

**Returns**

The `call_once` function returns no value.

### 7.26.3 Condition variable functions

#### 7.26.3.1 The `cnd_broadcast` function

**Synopsis**

```c
#include <threads.h>

int cnd_broadcast(cnd_t *cond);
```
**Description**

The `cnd_broadcast` function unblocks all of the threads that are blocked on the condition variable pointed to by `cond` at the time of the call. If no threads are blocked on the condition variable pointed to by `cond` at the time of the call, the function does nothing.

**Returns**

The `cnd_broadcast` function returns `thrd_success` on success, or `thrd_error` if the request could not be honored.

### 7.26.3.2 The `cnd_destroy` function

**Synopsis**

```c
#include <threads.h>

void cnd_destroy(cnd_t *cond);
```

**Description**

The `cnd_destroy` function releases all resources used by the condition variable pointed to by `cond`. The `cnd_destroy` function requires that no threads be blocked waiting for the condition variable pointed to by `cond`.

**Returns**

The `cnd_destroy` function returns no value.

### 7.26.3.3 The `cnd_init` function

**Synopsis**

```c
#include <threads.h>

int cnd_init(cnd_t *cond);
```

**Description**

The `cnd_init` function creates a condition variable. If it succeeds it sets the variable pointed to by `cond` to a value that uniquely identifies the newly created condition variable. A thread that calls `cnd_wait` on a newly created condition variable will block.

**Returns**

The `cnd_init` function returns `thrd_success` on success, or `thrd_nomem` if no memory could be allocated for the newly created condition, or `thrd_error` if the request could not be honored.

### 7.26.3.4 The `cnd_signal` function

**Synopsis**

```c
#include <threads.h>

int cnd_signal(cnd_t *cond);
```

**Description**

The `cnd_signal` function unblocks one of the threads that are blocked on the condition variable pointed to by `cond` at the time of the call. If no threads are blocked on the condition variable at the time of the call, the function does nothing and returns success.

**Returns**

The `cnd_signal` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

### 7.26.3.5 The `cnd_timedwait` function

**Synopsis**

```c
#include <threads.h>

int cnd_timedwait(cnd_t *cond, mtx_t *restrict mtx, const struct timespec *restrict ts);
```
Description
2 The \texttt{cnd\_timedwait} function atomically unlocks the mutex pointed to by \texttt{mtx} and blocks until the condition variable pointed to by \texttt{cond} is signaled by a call to \texttt{cnd\_signal} or to \texttt{cnd\_broadcast}, or until after the \texttt{TIME\_UTC}-based calendar time pointed to by \texttt{ts}, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the variable pointed to by \texttt{mtx} before it returns. The \texttt{cnd\_timedwait} function requires that the mutex pointed to by \texttt{mtx} be locked by the calling thread.

Returns
3 The \texttt{cnd\_timedwait} function returns \texttt{thrd\_success} upon success, or \texttt{thrd\_timedout} if the time specified in the call was reached without acquiring the requested resource, or \texttt{thrd\_error} if the request could not be honored.

\textbf{7.26.3.6 The \texttt{cnd\_wait} function}

Synopsis
1
\begin{verbatim}
#include <threads.h>
int cnd_wait(cnd_t *cond, mtx_t *mtx);
\end{verbatim}

Description
2 The \texttt{cnd\_wait} function atomically unlocks the mutex pointed to by \texttt{mtx} and blocks until the condition variable pointed to by \texttt{cond} is signaled by a call to \texttt{cnd\_signal} or to \texttt{cnd\_broadcast}, or until it is unblocked due to an unspecified reason. When the calling thread becomes unblocked it locks the mutex pointed to by \texttt{mtx} before it returns. The \texttt{cnd\_wait} function requires that the mutex pointed to by \texttt{mtx} be locked by the calling thread.

Returns
3 The \texttt{cnd\_wait} function returns \texttt{thrd\_success} on success or \texttt{thrd\_error} if the request could not be honored.

\textbf{7.26.4 Mutex functions}

For purposes of determining the existence of a data race, lock and unlock operations behave as atomic operations. All lock and unlock operations on a particular mutex occur in some particular total order.

\begin{multicols}{2}
2 \textbf{NOTE} This total order can be viewed as the modification order of the mutex.
\end{multicols}

\textbf{7.26.4.1 The \texttt{mtx\_destroy} function}

Synopsis
1
\begin{verbatim}
#include <threads.h>
void mtx_destroy(mtx_t *mtx);
\end{verbatim}

Description
2 The \texttt{mtx\_destroy} function releases any resources used by the mutex pointed to by \texttt{mtx}. No threads can be blocked waiting for the mutex pointed to by \texttt{mtx}.

Returns
3 The \texttt{mtx\_destroy} function returns no value.

\textbf{7.26.4.2 The \texttt{mtx\_init} function}

Synopsis
1
\begin{verbatim}
#include <threads.h>
int mtx_init(mtx_t *mtx, int type);
\end{verbatim}
Description

The `mtx_init` function creates a mutex object with properties indicated by `type`, which shall have one of these values:

- `mtx_plain` for a simple non-recursive mutex,
- `mtx_timed` for a non-recursive mutex that supports timeout,
- `mtx_plain | mtx_recursive` for a simple recursive mutex, or
- `mtx_timed | mtx_recursive` for a recursive mutex that supports timeout.

If the `mtx_init` function succeeds, it sets the mutex pointed to by `mtx` to a value that uniquely identifies the newly created mutex.

Returns

The `mtx_init` function returns `thrd_success` on success, or `thrd_error` if the request could not be honored.

7.26.4.3 The `mtx_lock` function

Synopsis

```c
#include <threads.h>
int mtx_lock (mtx_t *mtx);
```

Description

The `mtx_lock` function blocks until it locks the mutex pointed to by `mtx`. If the mutex is non-recursive, it shall not be locked by the calling thread. Prior calls to `mtx_unlock` on the same mutex synchronize with this operation.

Returns

The `mtx_lock` function returns `thrd_success` on success, or `thrd_error` if the request could not be honored.

7.26.4.4 The `mtx_timedlock` function

Synopsis

```c
#include <threads.h>
int mtx_timedlock (mtx_t *restrict mtx, const struct timespec *restrict ts);
```

Description

The `mtx_timedlock` function endeavors to block until it locks the mutex pointed to by `mtx` or until after the `TIME_UTC`-based calendar time pointed to by `ts`. The specified mutex shall support timeout. If the operation succeeds, prior calls to `mtx_unlock` on the same mutex synchronize with this operation.

Returns

The `mtx_timedlock` function returns `thrd_success` on success, or `thrd_timedout` if the time specified was reached without acquiring the requested resource, or `thrd_error` if the request could not be honored.

7.26.4.5 The `mtx_trylock` function

Synopsis

```c
#include <threads.h>
int mtx_trylock (mtx_t *mtx);
```
Description
2 The `mtx_trylock` function endeavors to lock the mutex pointed to by `mtx`. If the mutex is already locked, the function returns without blocking. If the operation succeeds, prior calls to `mtx_unlock` on the same mutex synchronize with this operation.

Returns
3 The `mtx_trylock` function returns `thrd_success` on success, or `thrd_busy` if the resource requested is already in use, or `thrd_error` if the request could not be honored. `mtx_trylock` may spuriously fail to lock an unused resource, in which case it returns `thrd_busy`.

7.26.4.6 The `mtx_unlock` function
Synopsis
1
```
#include <threads.h>
int mtx_unlock(mtx_t *mtx);
```

Description
2 The `mtx_unlock` function unlocks the mutex pointed to by `mtx`. The mutex pointed to by `mtx` shall be locked by the calling thread.

Returns
3 The `mtx_unlock` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

7.26.5 Thread functions
7.26.5.1 The `thrd_create` function
Synopsis
1
```
#include <threads.h>
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Description
2 The `thrd_create` function creates a new thread executing `func(arg)`. If the `thrd_create` function succeeds, it sets the object pointed to by `thr` to the identifier of the newly created thread. (A thread’s identifier may be reused for a different thread once the original thread has exited and either been detached or joined to another thread.) The completion of the `thrd_create` function synchronizes with the beginning of the execution of the new thread.

Returns
4 The `thrd_create` function returns `thrd_success` on success, or `thrd_nomem` if no memory could be allocated for the thread requested, or `thrd_error` if the request could not be honored.

7.26.5.2 The `thrd_current` function
Synopsis
1
```
#include <threads.h>
thrd_t thrd_current(void);
```

Description
2 The `thrd_current` function identifies the thread that called it.

Returns
3 The `thrd_current` function returns the identifier of the thread that called it.
7.26.5.3 The `thrd_detach` function

**Synopsis**

```c
#include <threads.h>
int thrd_detach(thrd_t thr);
```

**Description**

The `thrd_detach` function tells the operating system to dispose of any resources allocated to the thread identified by `thr` when that thread terminates. The thread identified by `thr` shall not have been previously detached or joined with another thread.

**Returns**

The `thrd_detach` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

7.26.5.4 The `thrd_equal` function

**Synopsis**

```c
#include <threads.h>
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

**Description**

The `thrd_equal` function will determine whether the thread identified by `thr0` refers to the thread identified by `thr1`.

**Returns**

The `thrd_equal` function returns zero if the thread `thr0` and the thread `thr1` refer to different threads. Otherwise the `thrd_equal` function returns a nonzero value.

7.26.5.5 The `thrd_exit` function

**Synopsis**

```c
#include <threads.h>
_Noreturn void thrd_exit(int res);
```

**Description**

For every thread-specific storage key which was created with a non-null destructor and for which the value is non-null, `thrd_exit` sets the value associated with the key to a null pointer value and then invokes the destructor with its previous value. The order in which destructors are invoked is unspecified.

If after this process there remain keys with both non-null destructors and values, the implementation repeats this process up to `TSS_DTOR_ITERATIONS` times.

Following this, the `thrd_exit` function terminates execution of the calling thread and sets its result code to `res`.

The program terminates normally after the last thread has been terminated. The behavior is as if the program called the `exit` function with the status `EXIT_SUCCESS` at thread termination time.

**Returns**

The `thrd_exit` function returns no value.

7.26.5.6 The `thrd_join` function

**Synopsis**

```c
#include <threads.h>
int thrd_join(thrd_t thr, int *res);
```
Description
2 The **thrd_join** function joins the thread identified by `thr` with the current thread by blocking until the other thread has terminated. If the parameter `res` is not a null pointer, it stores the thread’s result code in the integer pointed to by `res`. The termination of the other thread synchronizes with the completion of the **thrd_join** function. The thread identified by `thr` shall not have been previously detached or joined with another thread.

Returns
3 The **thrd_join** function returns **thrd_success** on success or **thrd_error** if the request could not be honored.

7.26.5.7 The **thrd_sleep** function
Synopsis
    ```
    #include <threads.h>
    int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
    ```

Description
2 The **thrd_sleep** function suspends execution of the calling thread until either the interval specified by `duration` has elapsed or a signal which is not being ignored is received. If interrupted by a signal and the `remaining` argument is not null, the amount of time remaining (the requested interval minus the time actually slept) is stored in the interval it points to. The `duration` and `remaining` arguments may point to the same object.

3 The suspension time may be longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than that specified, as measured by the system clock `TIME_UTC`.

Returns
4 The **thrd_sleep** function returns zero if the requested time has elapsed, −1 if it has been interrupted by a signal, or a negative value (which may also be −1) if it fails.

7.26.5.8 The **thrd_yield** function
Synopsis
    ```
    #include <threads.h>
    void thrd_yield(void);
    ```

Description
2 The **thrd_yield** function endeavors to permit other threads to run, even if the current thread would ordinarily continue to run.

Returns
3 The **thrd_yield** function returns no value.

7.26.6 Thread-specific storage functions
7.26.6.1 The **tss_create** function
Synopsis
    ```
    #include <threads.h>
    int tss_create(tss_t *key, tss_dtor_t dtor);
    ```

Description
2 The **tss_create** function creates a thread-specific storage pointer with destructor `dtor`, which may be null.
A null pointer value is associated with the newly created key in all existing threads. Upon subsequent thread creation, the value associated with all keys is initialized to a null pointer value in the new thread.

Destructors associated with thread-specific storage are not invoked at program termination.

The `tss_create` function shall not be called from within a destructor.

Returns

If the `tss_create` function is successful, it sets the thread-specific storage pointed to by key to a value that uniquely identifies the newly created pointer and returns `thrd_success`; otherwise, `thrd_error` is returned and the thread-specific storage pointed to by key is set to an indeterminate value.

### 7.26.6.2 The `tss_delete` function

**Synopsis**

```c
#include <threads.h>
void tss_delete(tss_t key);
```

**Description**

The `tss_delete` function releases any resources used by the thread-specific storage identified by key. The `tss_delete` function shall only be called with a value for key that was returned by a call to `tss_create` before the thread commenced executing destructors.

If `tss_delete` is called while another thread is executing destructors, whether this will affect the number of invocations of the destructor associated with key on that thread is unspecified.

Calling `tss_delete` will not result in the invocation of any destructors.

Returns

The `tss_delete` function returns no value.

### 7.26.6.3 The `tss_get` function

**Synopsis**

```c
#include <threads.h>
void *tss_get(tss_t key);
```

**Description**

The `tss_get` function returns the value for the current thread held in the thread-specific storage identified by key. The `tss_get` function shall only be called with a value for key that was returned by a call to `tss_create` before the thread commenced executing destructors.

Returns

The `tss_get` function returns the value for the current thread if successful, or zero if unsuccessful.

### 7.26.6.4 The `tss_set` function

**Synopsis**

```c
#include <threads.h>
int tss_set(tss_t key, void *val);
```

**Description**

The `tss_set` function sets the value for the current thread held in the thread-specific storage identified by key to val. The `tss_set` function shall only be called with a value for key that was returned by a call to `tss_create` before the thread commenced executing destructors.

This action will not invoke the destructor associated with the key on the value being replaced.
Returns

4 The `tss_set` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.
7.27 Date and time <time.h>

7.27.1 Components of time

The header <time.h> defines several macros, and declares types and functions for manipulating time. Many functions deal with a calendar time that represents the current date (according to the Gregorian calendar) and time. Some functions deal with local time, which is the calendar time expressed for some specific time zone, and with Daylight Saving Time, which is a temporary change in the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

The feature test macro __STDC_VERSION_TIME_H__ expands to the token yyyymm. The other macros defined are NULL (described in 7.19);

```
CLOCKS_PER_SEC
```

which expands to an expression with type clock_t (described below) that is the number per second of the value returned by the clock function; and

```
TIME_UTC
```

which expands to an integer constant greater than 0 that designates the UTC time base.337)

The types declared are size_t (described in 7.19);

```
clock_t
```

and

```
time_t
```

which are real types capable of representing times;

```
struct timespec
```

which holds an interval specified in seconds and nanoseconds (which may represent a calendar time based on a particular epoch); and

```
struct tm
```

which holds the components of a calendar time, called the broken-down time.

The range and precision of times representable in clock_t and time_t are implementation-defined. The timespec structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.338)

```
time_t tv_sec; // whole seconds -- ≥ 0
long tv_nsec; // nanoseconds -- [0, 999999999]
```

The tm structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are expressed in the comments.339)

```
int tm_sec; // seconds after the minute -- [0, 60]
int tm_min; // minutes after the hour -- [0, 59]
int tm_hour; // hours since midnight -- [0, 23]
int tm_mday; // day of the month -- [1, 31]
int tm_mon; // months since January -- [0, 11]
```

337) Implementations can define additional time bases, but are only required to support a real time clock based on UTC.
338) The tv_sec member is a linear count of seconds and might not have the normal semantics of a time_t.
339) The range [0, 60] for tm_sec allows for a positive leap second.
int tm_year; // years since 1900
int tm_wday; // days since Sunday -- [0, 6]
int tm_yday; // days since January 1 -- [0, 365]
int tm_isdst; // Daylight Saving Time flag

The value of tm_isdst is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

7.27.2 Time manipulation functions
7.27.2.1 The clock function
Synopsis

#include <time.h>
clock_t clock(void);

Description
The clock function determines the processor time used.

Returns
The clock function returns the implementation’s best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the clock function should be divided by the value of the macro CLSX_PER_SEC. If the processor time used is not available, the function returns the value (clock_t)(−1). If the value cannot be represented, the function returns an unspecified value.340

7.27.2.2 The difftime function
Synopsis

#include <time.h>
double difftime(time_t timel, time_t time0);

Description
The difftime function computes the difference between two calendar times: timel - time0.

Returns
The difftime function returns the difference expressed as a double.

7.27.2.3 The mktime function
Synopsis

#include <time.h>

time_t mktime(struct tm *timeptr);

Description
The mktime function converts the broken-down time, expressed as local time, in the structure pointed to by timeptr into a calendar time value with the same encoding as that of the values returned by the time function. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above.341 On successful completion, the values of the tm_wday and tm_yday components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of tm_mday is not set until tm_mon and tm_year are determined.

340) This could be due to overflow of the clock_t type.
341) Thus, a positive or zero value for tm_isdst causes the mktime function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value causes it to attempt to determine whether Daylight Saving Time is in effect for the specified time.
Returns
3 The \texttt{mktime} function returns the specified calendar time encoded as a value of type \texttt{time_t}. If the calendar time cannot be represented, the function returns the value \((\texttt{time_t})(-1))\.

\begin{verbatim}
#include <stdio.h>
#include <time.h>

static const char *const wday[] = {
   "Sunday", "Monday", "Tuesday", "Wednesday",
   "Thursday", "Friday", "Saturday", "-unknown-
};

struct tm time_str;
  /* ... */

time_str.tm_year = 2001 - 1900;
time_str.tm_mon  = 7 - 1;
time_str.tm_mday = 4;
time_str.tm_hour = 0;
time_str.tm_min  = 0;
time_str.tm_sec  = 1;
time_str.tm_isdst = -1;
if (mktime(&time_str) == (time_t)(-1))
   time_str.tm_wday = 7;
printf("%s
", wday[time_str.tm_wday]);
\end{verbatim}

7.27.2.4 The \texttt{time} function

Synopsis
1
\begin{verbatim}
#include <time.h>

time_t time(time_t *timer);
\end{verbatim}

Description
2 The \texttt{time} function determines the current calendar time. The encoding of the value is unspecified.

Returns
3 The \texttt{time} function returns the implementation’s best approximation to the current calendar time. The value \((\texttt{time_t})(-1))\ is returned if the calendar time is not available. If \texttt{timer} is not a null pointer, the return value is also assigned to the object it points to.

7.27.2.5 The \texttt{timespec_get} function

Synopsis
1
\begin{verbatim}
#include <time.h>

int timespec_get(struct timespec *ts, int base);
\end{verbatim}

Description
2 The \texttt{timespec_get} function sets the interval pointed to by \texttt{ts} to hold the current calendar time based on the specified time base.

3 If \texttt{base} is \texttt{TIME_UTC}, the \texttt{tv_sec} member is set to the number of seconds since an implementation defined \textit{epoch}, truncated to a whole value and the \texttt{tv_nsec} member is set to the integral number of nanoseconds, rounded to the resolution of the system clock.\footnote{Although a \texttt{struct timespec} object describes times with nanosecond resolution, the available resolution is system dependent and could even be greater than 1 second.}

Returns
4 If the \texttt{timespec_get} function is successful it returns the nonzero value \texttt{base}; otherwise, it returns zero.
7.27.3 Time conversion functions

Except pointed types the same a them two the of:

any objects:
structure converts function and object in of the:
to by of:::

returned from any previous value of object for any the

The

ranges,
using the equivalent of the following algorithm.

3
If any of the members of the broken-down time contain values that are outside their normal ranges, the behavior of the asctime functions is undefined. Likewise, if the calculated year exceeds four digits or is less than the year 1000, the behavior is undefined. The buf parameter for asctime_r shall point to a buffer of at least 26 bytes.

\[3^{43}\]This does not mean that these functions may not read global state that describes the time and calendar settings of the execution, such as the LC_TIME locale or the implementation defined specification of the local time zone. Only the setting of that state by setlocale or by means of implementation-defined functions may constitute races.

\[3^{44}\]See 7.27.1.
Returns

The `asctime` functions return a pointer to the string.

7.27.3.2 The `ctime` functions

Synopsis replace Synopsis

```c
#include <time.h>
char *ctime(const time_t *timer);
char *ctime_r(const time_t *timer, char *buf);
```

Description

The `ctime` functions convert the calendar time pointed to by `timer` to local time in the form of a string. They are equivalent to:

```c
asctime(localtime_r(timer, (struct tm[1]){ 0 }))
```

and

```c
asctime_r(localtime_r(timer, (struct tm[1]){ 0 }), buf)
```

The `buf` parameter for `ctime_r` shall point to a buffer of at least 26 bytes.

Returns

The `ctime` functions return the pointer returned by the `asctime` function with that broken-down time as argument.

Forward references: the `localtime` function (§7.27.3.4).

7.27.3.3 The `gmtime` functions

Synopsis replace Synopsis

```c
#include <time.h>
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *timer, struct tm *buf);
```

Description

The `gmtime` functions convert the calendar time pointed to by `timer` into a broken-down time, expressed as UTC.

Returns

The `gmtime` functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to UTC.

7.27.3.4 The `localtime` functions

Synopsis replace Synopsis

```c
#include <time.h>
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *timer, struct tm *buf);
```

Description

The `localtime` functions converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time.

Returns

The `localtime` functions return a pointer to the broken-down time, or a null pointer if the specified time cannot be converted to local time.

7.27.3.5 The `strftime` function
# Synopsis

```c
#include <time.h>

size_t strftime(char * restrict s, size_t maxsize, const char * restrict format, const struct tm * restrict timeptr);
```

## Description

The `strftime` function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. The format string shall be a multibyte character sequence, beginning and ending in its initial shift state. The `format` string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a `%` character, possibly followed by an `E` or `O` modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array.

Each conversion specifier shall be replaced by appropriate characters as described in the following list. The appropriate characters shall be determined using the `LC_TIME` category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by `timeptr`, as specified in brackets in the description. If any of the specified values is outside the normal range, the characters stored are unspecified.

- `%a` is replaced by the locale’s abbreviated weekday name.  
  [tm_wday]
- `%A` is replaced by the locale’s full weekday name.  
  [tm_wday]
- `%b` is replaced by the locale’s abbreviated month name.  
  [tm_mon]
- `%B` is replaced by the locale’s full month name.  
  [tm_mon]
- `%c` is replaced by the locale’s appropriate date and time representation.  
  [all specified in 7.27.1]
- `%C` is replaced by the year divided by 100 and truncated to an integer, as a decimal number (00–99).  
  [tm_year]
- `%d` is replaced by the day of the month as a decimal number (01–31).  
  [tm_mday]
- `%D` is equivalent to “%m/%d/%y”.  
  [tm_mon, tm_mday, tm_year]
- `%e` is replaced by the day of the month as a decimal number (1–31); a single digit is preceded by a space.  
  [tm_mday]
- `%f` is equivalent to “%Y-%m-%d” (the ISO 8601 date format).  
  [tm_year, tm_mon, tm_mday]
- `%g` is replaced by the last 2 digits of the week-based year (see below) as a decimal number (00–99).  
  [tm_year, tm_wday, tm_yday]
- `%G` is replaced by the week-based year (see below) as a decimal number (e.g., 1997).  
  [tm_year, tm_wday, tm_yday]
- `%h` is equivalent to “%b”.  
  [tm_mon]
- `%H` is replaced by the hour (24-hour clock) as a decimal number (00–23).  
  [tm_hour]
- `%I` is replaced by the hour (12-hour clock) as a decimal number (01–12).  
  [tm_hour]
- `%j` is replaced by the day of the year as a decimal number (001–366).  
  [tm_yday]
- `%m` is replaced by the month as a decimal number (01–12).  
  [tm_mon]
- `%M` is replaced by the minute as a decimal number (00–59).  
  [tm_min]
- `%n` is replaced by a new-line character.
- `%p` is replaced by the locale’s equivalent of the AM/PM designations associated with a 12-hour clock.  
  [tm_hour]
- `%r` is replaced by the locale’s 12-hour clock time.  
  [tm_hour, tm_min, tm_sec]
- `%R` is equivalent to “%H:%M”.  
  [tm_hour, tm_min]
- `%S` is replaced by the second as a decimal number (00–60).  
  [tm_sec]
%t is replaced by a horizontal-tab character.

%T is equivalent to “%H:%M:%S” (the ISO 8601 time format). [tm_hour, tm_min, tm_sec]

%u is replaced by the ISO 8601 weekday as a decimal number (1–7), where Monday is 1. [tm_wday]

%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00–53). [tm_year, tm_wday, tm_yday]

%V is replaced by the ISO 8601 week number (see below) as a decimal number (01–53). [tm_year, tm_wday, tm_yday]

%w is replaced by the weekday as a decimal number (0–6), where Sunday is 0. [tm_wday]

%W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (00–53). [tm_year, tm_wday, tm_yday]

%x is replaced by the locale’s appropriate date representation. [all specified in 7.27.1]

%X is replaced by the locale’s appropriate time representation. [all specified in 7.27.1]

%y is replaced by the last 2 digits of the year as a decimal number (00–99). [tm_year]

%Y is replaced by the year as a decimal number (e.g., 1997). [tm_year]

%z is replaced by the offset from UTC in the ISO 8601 format “-0430” (meaning 4 hours 30 minutes behind UTC, west of Greenwich), or by no characters if no time zone is determinable. [tm_isdst]

%Z is replaced by the locale’s time zone name or abbreviation, or by no characters if no time zone is determinable. [tm_isdst]

%% is replaced by %.

---

Some conversion specifiers can be modified by the inclusion of an E or O modifier character to indicate an alternative format or specification. If the alternative format or specification does not exist for the current locale, the modifier is ignored.

%Ec is replaced by the locale’s alternative date and time representation.

%EC is replaced by the name of the base year (period) in the locale’s alternative representation.

%Ex is replaced by the locale’s alternative date representation.

%EX is replaced by the locale’s alternative time representation.

%Ey is replaced by the offset from %EC (year only) in the locale’s alternative representation.

%Ey is replaced by the locale’s full alternative year representation.

%Ob is replaced by the locale’s abbreviated alternative month name.

%OB is replaced by the locale’s alternative appropriate full month name.

%Od is replaced by the day of the month, using the locale’s alternative numeric symbols (filled as needed with leading zeros, or with leading spaces if there is no alternative symbol for zero).

%Oe is replaced by the day of the month, using the locale’s alternative numeric symbols (filled as needed with leading spaces).

%OH is replaced by the hour (24-hour clock), using the locale’s alternative numeric symbols.

%OI is replaced by the hour (12-hour clock), using the locale’s alternative numeric symbols.

%0m is replaced by the month, using the locale’s alternative numeric symbols.

%OM is replaced by the minutes, using the locale’s alternative numeric symbols.

%OS is replaced by the seconds, using the locale’s alternative numeric symbols.

%ou is replaced by the ISO 8601 weekday as a number in the locale’s alternative representation, where Monday is 1.

%0U is replaced by the week number, using the locale’s alternative numeric symbols.

%0V is replaced by the ISO 8601 week number, using the locale’s alternative numeric symbols.

%0w is replaced by the weekday as a number, using the locale’s alternative numeric symbols.
%OW is replaced by the week number of the year, using the locale’s alternative numeric symbols.
%Oy is replaced by the last 2 digits of the year, using the locale’s alternative numeric symbols.

5 %g, %G, and %V give values according to the ISO 8601 week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, %G is replaced by 1998 and %V is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, %G is replaced by 1998 and %V is replaced by 01.

6 If a conversion specifier is not one of the above, the behavior is undefined.

7 In the “C” locale, the E and O modifiers are ignored and the replacement strings for the following specifiers are:

   %a the first three characters of %A.
   %A one of “Sunday”, “Monday”, . . . , “Saturday”.
   %b the first three characters of %B.
   %B one of “January”, “February”, . . . , “December”.
   %c equivalent to “%a %b %e %T %Y”.
   %p one of “AM” or “PM”.
   %r equivalent to “%I:%M:%S %p”.
   %x equivalent to “%m/%d/%y”.
   %X equivalent to %T.
   %Z implementation-defined.

Returns

8 If the total number of resulting characters including the terminating null character is not more than maxsize, the strftime function returns the number of characters placed into the array pointed to by s not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.
7.28 Unicode utilities <uchar.h>

The header <uchar.h> declares types and functions for manipulating Unicode characters.

The types declared are `mbstate_t` (described in 7.29.1) and `size_t` (described in 7.19);

```
char16_t
```

which is an unsigned integer type used for 16-bit characters and is the same type as `uint_least16_t` (described in 7.20.1.2); and

```
char32_t
```

which is an unsigned integer type used for 32-bit characters and is the same type as `uint_least32_t` (also described in 7.20.1.2).

7.28.1 Restartable multibyte/wide character conversion functions

These functions have a parameter, `ps`, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence, which the functions alter as necessary. If `ps` is a null pointer, each function uses its own internal `mbstate_t` object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

7.28.1.1 The `mbrtoc16` function

Synopsis

```
#include <uchar.h>
size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

If `s` is a null pointer, the `mbrtoc16` function is equivalent to the call:

```
mbrtoc16(NULL, ",", 1, ps)
```

In this case, the values of the parameters `pc16` and `n` are ignored.

If `s` is not a null pointer, the `mbrtoc16` function inspects at most `n` bytes beginning with the byte pointed to by `s` to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if `pc16` is not a null pointer, stores the value of the first (or only) such character in the object pointed to by `pc16`. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The `mbrtoc16` function returns the first of the following that applies (given the current conversion state):

- `0` if the next `n` or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).
- `between 1 and n inclusive` if the next `n` or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.
(size_t)(−3) if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(size_t)(−2) if the next n bytes contribute to an incomplete (but potentially valid) multibyte character, and all n bytes have been processed (no value is stored).

(size_t)(−1) if an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro EILSEQ is stored in errno, and the conversion state is unspecified.

7.28.1.2 The c16rtomb function

Synopsis

```c
#include <uchar.h>
size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
```

Description

If s is a null pointer, the c16rtomb function is equivalent to the call

```c
    c16rtomb(buf, L'\0', ps)
```

where buf is an internal buffer.

If s is not a null pointer, the c16rtomb function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given or completed by c16 (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by s, or stores nothing if c16 does not represent a complete character. At most MB_CUR_MAX bytes are stored. If c16 is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

The c16rtomb function returns the number of bytes stored in the array object (including any shift sequences). When c16 is not a valid wide character, an encoding error occurs: the function stores the value of the macro EILSEQ in errno and returns (size_t)(−1); the conversion state is unspecified.

7.28.1.3 The mbrtoc32 function

Synopsis

```c
#include <uchar.h>
size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

If s is a null pointer, the mbrtoc32 function is equivalent to the call:

```c
    mbrtoc32(NULL, "", 1, ps)
```

In this case, the values of the parameters pc32 and n are ignored.

If s is not a null pointer, the mbrtoc32 function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if pc32 is not a null pointer, stores the value of the first (or only) such character in the object pointed to by pc32. Subsequent calls will store successive wide characters without consuming any additional

---

345) When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The \texttt{mbrtoc32} function returns the first of the following that applies (given the current conversion state):

- \texttt{0} if the next \texttt{n} or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).
- \texttt{between 1 and n inclusive} if the next \texttt{n} or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.
- \texttt{(size_t)(-3)} if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).
- \texttt{(size_t)(-2)} if the next \texttt{n} bytes contribute to an incomplete (but potentially valid) multibyte character, and all \texttt{n} bytes have been processed (no value is stored).\footnote{When \texttt{n} has at least the value of the \texttt{MB_CUR_MAX} macro, this case can only occur if \texttt{s} points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).}
- \texttt{(size_t)(-1)} if an encoding error occurs, in which case the next \texttt{n} or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro \texttt{EILSEQ} is stored in \texttt{errno}, and the conversion state is unspecified.

7.28.1.4 The \texttt{c32rtomb} function

Synopsis

\begin{verbatim}
#include <uchar.h>
size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
\end{verbatim}

Description

If \texttt{s} is a null pointer, the \texttt{c32rtomb} function is equivalent to the call

\begin{verbatim}
c32rtomb(buf, L'\0', ps)
\end{verbatim}

where \texttt{buf} is an internal buffer.

If \texttt{s} is not a null pointer, the \texttt{c32rtomb} function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by \texttt{c32} (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by \texttt{s}. At most \texttt{MB_CUR_MAX} bytes are stored. If \texttt{c32} is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

The \texttt{c32rtomb} function returns the number of bytes stored in the array object (including any shift sequences). When \texttt{c32} is not a valid wide character, an encoding error occurs: the function stores the value of the macro \texttt{EILSEQ} in \texttt{errno} and returns \texttt{(size_t)(-1)}; the conversion state is unspecified.
7.29 Extended multibyte and wide character utilities <wchar.h>

7.29.1 Introduction

The header <wchar.h> defines four macros, and declares four data types, one tag, and many functions.\(^\text{347}\) The types declared are `wchar_t` and `size_t` (both described in 7.19);

\[
\text{mbstate_t}
\]

which is a complete object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

\[
wint_t
\]

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see \textit{WEOF} below);\(^\text{348}\) and

\[
\text{struct tm}
\]

which is declared as an incomplete structure type (the contents are described in 7.27.1).

The macros defined are \textit{NULL} (described in 7.19); `\text{WCHAR_MIN}` and `\text{WCHAR_MAX}` (described in 7.20.3); and

\[
\text{WEOF}
\]

which expands to a constant expression of type `\text{wint_t}` whose value does not correspond to any member of the extended character set.\(^\text{349}\) It is accepted (and returned) by several functions in this subclause to indicate \textit{end-of-file}, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

The functions declared are grouped as follows:

- Functions that perform input and output of wide characters, or multibyte characters, or both;
- Functions that provide wide string numeric conversion;
- Functions that perform general wide string manipulation;
- Functions for wide string date and time conversion; and
- Functions that provide extended capabilities for conversion between multibyte and wide character sequences.

Arguments to the functions in this subclause may point to arrays containing `wchar_t` values that do not correspond to members of the extended character set. Such values shall be processed according to the specified semantics, except that it is unspecified whether an encoding error occurs if such a value appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by `\text{wcrtomb}`.

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

\(^\text{347}\)See “future library directions” (7.31.18).

\(^\text{348}\) `\text{wchar_t}` and `\text{wint_t}` can be the same integer type.

\(^\text{349}\) The value of the macro `\text{WEOF}` can differ from that of `\text{EIF}` and need not be negative.
7.29.2 Formatted wide character input/output functions

The formatted wide character input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.\(^{350}\)

7.29.2.1 The \texttt{fwprintf} function

\texttt{Synopsis replace Synopsis}

\begin{verbatim}
#include <stdio.h>
#include <wchar.h>

int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
\end{verbatim}

\textbf{Description}

The \texttt{fwprintf} function writes output to the stream pointed to by \texttt{stream}, under control of the wide string pointed to by \texttt{format} that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The \texttt{fwprintf} function returns when the end of the format string is encountered.

The format is composed of zero or more directives: ordinary wide characters (not \%), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the wide character \%\%. After the \%, the following appear in sequence:

\begin{itemize}
\item Zero or more flags (in any order) that modify the meaning of the conversion specification.
\item An optional minimum \textbf{field width}. If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a nonnegative decimal integer.\(^{351}\)
\item An optional \textbf{precision} that gives the minimum number of digits to appear for the \texttt{d}, \texttt{i}, \texttt{o}, \texttt{u}, \texttt{x}, and \texttt{X} conversions, the number of digits to appear after the decimal-point wide character for \texttt{a}, \texttt{A}, \texttt{e}, \texttt{E}, \texttt{f}, and \texttt{F} conversions, the maximum number of significant digits for the \texttt{g} and \texttt{G} conversions, or the maximum number of wide characters to be written for \texttt{s} conversions. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional nonnegative decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
\item An optional \textbf{length modifier} that specifies the size of the argument.
\item A \textbf{conversion specifier} wide character that specifies the type of conversion to be applied.
\end{itemize}

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an \texttt{int} argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a \texttt{-} flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag wide characters and their meanings are:

\begin{itemize}
\item The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
\item The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)\(^{352}\)
\end{itemize}

\(^{350}\)The \texttt{fwprintf} functions perform writes to memory for the \texttt{%n} specifier.
\(^{351}\)Note that 0 is taken as a flag, not as the beginning of a field width.
\(^{352}\)The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.
The length modifiers and their meanings are:

- `hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.
- `h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short int` or `unsigned short int` before printing); or that a following `n` conversion specifier applies to a pointer to a `short int` argument.
- `l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; that a following `c` conversion specifier applies to a `wchar_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `E`, `f`, `F`, `g`, or `G` conversion specifier.
- `ll` (ell-ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long int` or `unsigned long long int` argument; or that a following `n` conversion specifier applies to a pointer to a `long long int` argument.
- `j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument.
- `z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.
- `t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.
- `L` Specifies that a following `a`, `A`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.
- `H` Specifies that a following `a`, `A`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `_Decimal32` argument.
The conversion specifiers and their meanings are:

- **d, i** The int argument is converted to signed decimal in the style [%]-dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

- **o, u, x, X** The unsigned int argument is converted to unsigned octal (0), unsigned decimal (u), or unsigned hexadecimal notation (x or X) in the style ddd; the letters ABCDEF are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

- **f, F** A double argument representing a floating-point number is converted to decimal notation in the style [%]-[dD]dd.d, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A double argument representing an infinity is converted in one of the styles [%]-[iI]nf or [%]-[iI]nfinity — which style is implementation-defined. A double argument representing a NaN is converted in one of the styles [%]-[nN]an or [%]-[nN]an(n-wchar-sequence) — which style, and the meaning of any n-wchar-sequence, is implementation-defined. The F conversion specifier produces INF, INFINITY, or NAN instead of inf, infinity, or nan, respectively.\(^{353}\)

- **e, E** A double argument representing a floating-point number is converted in the style [%]-[dD]d[deE]dd, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point wide character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

- **g, G** A double argument representing a floating-point number is converted in style f or e (or in style F or E in the case of a G conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style E would have an exponent of X:

  - if \( P > X \geq -4 \), the conversion is with style f (or F) and precision \( P - (X + 1) \).
  - otherwise, the conversion is with style e (or E) and precision \( P - 1 \).

Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point wide character is removed if there is no fractional portion remaining.

\(^{353}\)When applied to infinite and NaN values, the -, +, and space flag wide characters have their usual meaning; the # and 0 flag wide characters have no effect.
A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

A double argument representing a floating-point number is converted in the style \[ l-]0xh.lliilip±d, \] where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character 354 and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision is sufficient to distinguish 355 values of type double, except that trailing zeros may be omitted; if the precision is zero and the # flag is not specified, no decimal-point wide character appears. The letters abcdef are used for a conversion and the letters ABCDEF for a conversion. The A conversion specifier produces a number with X and P instead of x and p. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

If an H, D, or DD modifier is present and the precision is missing, then for a decimal floating type argument represented by a triple of integers (s, c, q), where n is the number of significant digits in the coefficient \( c \).

- if \( -(n + 5) \leq q \leq 0 \), use style f (or style F in the case of an A conversion specifier) with formatting precision equal to -q.
- otherwise, use style e (or style E in the case of an A conversion specifier) with formatting precision equal to \( n - 1 \), with the exceptions that if \( c = 0 \) then the digit-sequence in the exponent-part shall have the value q (rather than 0), and that the exponent is always expressed with the minimum number of digits required to represent its value (the exponent never contains a leading zero).

If the precision \( P \) is present (in the conversion specification) and is zero or at least as large as the precision \( p \) (5.2.4.2.2) of the decimal floating type, the conversion is as if the precision were missing. If the precision \( P \) is present (and nonzero) and less than the precision \( p \) of the decimal floating type, the conversion first obtains an intermediate result as follows, where \( n \) is the number of significant digits in the coefficient:

- If \( n \leq P \), set the intermediate result to the input.
- If \( n > P \), round the input value, according to the current rounding direction for decimal floating-point operations, to \( P \) decimal digits, with unbounded exponent range, representing the result with a \( P \)-digit integer coefficient when in the form \( (s, c, q) \).

Convert the intermediate result in the manner described above for the case where the precision is missing.

If no l length modifier is present, the int argument is converted to a wide character as if by calling bintowc and the resulting wide character is written.

If an l length modifier is present, the wint_t argument is converted to wchar_t and written.

If no l length modifier is present, the argument shall be a pointer to the initial element of a character array containing a multibyte character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the mbtowc

354 Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries.

355 The precision \( p \) is sufficient to distinguish values of the source type if \( 16^{p-1} > b^n \) where \( b \) is FLT_RADIX and \( n \) is the number of base-\( b \) digits in the significand of the source type. A smaller \( p \) might suffice depending on the implementation’s scheme for determining the digit to the left of the decimal-point wide character.
function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of `wchar_t` type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

p

The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printing wide characters, in an implementation-defined manner.

n

The argument shall be a pointer to signed integer into which is written the number of wide characters written to the output stream so far by this call to `fwprintf`. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A `%` wide character is written. No argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For `a` and `A` conversions, if `FLT_RADIX` is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

For `a` and `A` conversions, if `FLT_RADIX` is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most the maximum value $M$ of the `T_DECIMAL_DIG` macros (defined in `<float.h>`), then the result should be correctly rounded. If the number of significant decimal digits is more than `T_DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, the value of the resultant decimal string $D$ should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

The `fwprintf` function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

The number of wide characters that can be produced by any single conversion shall be at least 4095.

EXAMPLE To print a date and time in the form “Sunday, July 3, 10:02” followed by π to five decimal places:

```c
#include <math.h>
#include <stdio.h>
```

356 See “future library directions” (7.31.18).

357 For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.
#include <wchar.h>
/* ... */
wchar_t *weekday, *month;  // pointers to wide strings
int day, hour, min;
fwprintf(stdout, L"%ls, %ls %d, %.2d:%.2d\n",
    weekday, month, day, hour, min);
fwprintf(stdout, L"pi = %.5f\n", 4 * atan(1.0));

Forward references: the btowc function (7.29.6.1.), the mbtowc function (7.29.6.3.2.).

7.29.2.2 The fwscanf function

Synopsis

```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Description

The fwscanf function reads input from the stream pointed to by stream, under control of the wide string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format is composed of zero or more directives: one or more white-space wide characters, an ordinary wide character (neither % nor a white-space wide character), or a conversion specification. Each conversion specification is introduced by the wide character %. After the %, the following appear in sequence:

- An optional assignment-suppressing wide character *.
- An optional decimal integer greater than zero that specifies the maximum field width (in wide characters).
- An optional length modifier that specifies the size of the receiving object.
- A conversion specifier wide character that specifies the type of conversion to be applied.

The fwscanf function executes each directive of the format in turn. When all directives have been executed, or if a directive fails (as detailed below), the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read. The directive never fails.

A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If that wide character differs from the directive, the directive fails and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

1. Input white-space wide characters (as specified by the function) are skipped, unless the specification includes a [], c, or n specifier.\(^{358}\)
2. An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input wide characters which does not exceed any specified

\(^{358}\)These white-space wide characters are not counted against a specified field width.
The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %n directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

- hh Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to signed char or unsigned char.
- h Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to short int or unsigned short int.
- l (ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long int or unsigned long int; that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to double; or that a following c, s, or \[ conversion specifier applies to an argument with type pointer to wchar_t.
- ll (ell-ell) Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long long int or unsigned long long int.
- j Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t.
- z Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.
- t Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned integer type.
- L Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to long double.
- H Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal32.
- D Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal64.
- DD Specifies that a following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal128.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

- d Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the wcstol function with the value 10 for the base argument. The corresponding argument shall be a pointer to signed integer.

\[ fwscanf \] pushes back at most one input wide character onto the input stream. Therefore, some sequences that are acceptable to wcstod, wcstol, etc., are unacceptable to fwscanf.
Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the *wcstol* function with the value 0 for the *base* argument. The corresponding argument shall be a pointer to signed integer.

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the *wcstoul* function with the value 8 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *wcstoul* function with the value 10 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the *wcstoul* function with the value 16 for the *base* argument. The corresponding argument shall be a pointer to unsigned integer.

Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the *wcstod* function. The corresponding argument shall be a pointer to floating.

Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the directive).

If no *l* length modifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb* function, with the conversion state described by an *mbstate_t* object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an *l* length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of *wchar_t* large enough to accept the sequence. No null wide character is added.

Matches a sequence of non-white-space wide characters.

If no *l* length modifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb* function, with the conversion state described by an *mbstate_t* object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an *l* length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of *wchar_t* large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Matches a nonempty sequence of wide characters from a set of expected characters (the *scanset*).

If no *l* length modifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb* function, with the conversion state described by an *mbstate_t* object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an *l* length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of *wchar_t* large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the *format* string, up to and including the matching right bracket (*\*`). The wide characters between the brackets (the *scanset*) compose the scanset, unless the wide character after the left bracket is a circumflex (*\^*), in which case the scanset contains all wide characters that do not
appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with \[ \] or \[^\], the right bracket wide character is in the scanlist and the next following right bracket wide character is the matching right bracket that ends the specification; otherwise the first following right bracket wide character is the one that ends the specification. If a \(-\) wide character is in the scanlist and is not the first, nor the second where the first wide character is a \(^\), nor the last character, the behavior is implementation-defined.

\p \ Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the \%p conversion of the \texttt{fprintf} function. The corresponding argument shall be a pointer to a pointer to \texttt{void}. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the \%p conversion is undefined.

\n \ No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of wide characters read from the input stream so far by this call to the \texttt{fscanf} function. Execution of a \%n directive does not increment the assignment count returned at the completion of execution of the \texttt{fscanf} function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined.

\% \ Matches a single \% wide character; no conversion or assignment occurs. The complete conversion specification shall be \%%.

If a conversion specification is invalid, the behavior is undefined.\(^{360}\)

The conversion specifiers \(A\), \(E\), \(F\), \(G\), and \(X\) are also valid and behave the same as, respectively, \(a\), \(e\), \(f\), \(g\), and \(x\).

Trailing \texttt{white-space} wide characters (including new-line wide characters) are left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the \%n directive.

**Returns**

The \texttt{fscanf} function returns the value of the macro \texttt{EOF} if an input failure occurs before the first conversion (if any) has completed. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

---

\(^{360}\)See “future library directions” (7.31.18).
EXAMPLE 1 The call:

```c
#include <stdio.h>
#include <wchar.h>
/* ... */
int n, i; float x; wchar_t name[50];
 n = fwscanf(stdin, L"%d%f%ls", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to \( n \) the value 3, to \( i \) the value 25, to \( x \) the value 5.432, and to \( \text{name} \) the sequence \text{thompson}\0.

EXAMPLE 2 The call:

```c
#include <stdio.h>
#include <wchar.h>
/* ... */
i; float x; double y;
 fwscanf(stdin, L"%2d%f%*d %lf", &i, &x, &y);
```

with input:

```
56789 0123 56a72
```

will assign to \( i \) the value 56 and to \( x \) the value 789.0, will skip past 0123, and will assign to \( y \) the value 56.0. The next wide character read from the input stream will be a.

Forward references: the \text{wcstod}, \text{wcstof}, and \text{wcstold} functions (7.29.4.1.1), the \text{wcstol}, \text{wcstoll}, \text{wcstoul}, and \text{wcstoull} functions (7.29.4.1.3), the \text{wcrtomb} function (7.29.6.3.3).

7.29.2.3 The \text{swprintf} function

Synopsis

```c
#include <wchar.h>
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
```

Description

The \text{swprintf} function is equivalent to \text{fwprintf}, except that the argument \( s \) specifies an array of wide characters into which the generated output is to be written, rather than written to a stream. No more than \( n \) wide characters are written, including a terminating null wide character, which is always added (unless \( n \) is zero).

Returns

The \text{swprintf} function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if \( n \) or more wide characters were requested to be written.

7.29.2.4 The \text{swscanf} function

Synopsis

```c
#include <wchar.h>
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Description

The \text{swscanf} function is equivalent to \text{fwscanf}, except that the argument \( s \) specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the \text{fwscanf} function.
3 The `swscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `swscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

### 7.29.2.5 The `vfwprintf` function

#### Synopsis

```c
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

#### Description

2 The `vfwprintf` function is equivalent to `fwprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwprintf` function does not invoke the `va_end` macro.\(^\text{361}\)

#### Returns

3 The `vfwprintf` function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.  

4 **EXAMPLE** The following shows the use of the `vfwprintf` function in a general error-reporting routine.

```c
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

void error(char *function_name, wchar_t *format, ...) {
    va_list args;
    va_start(args, format);
    // print out name of function causing error
    fwprintf(stderr, L"ERROR in %s: ", function_name);
    // print out remainder of message
    vfwprintf(stderr, format, args);
    va_end(args);
}
```

### 7.29.2.6 The `vfwscanf` function

#### Synopsis

```c
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

#### Description

2 The `vfwscanf` function is equivalent to `fscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwscanf` function does not invoke the `va_end` macro.\(^\text{361}\)

\(^{361}\)As the functions `vfwprintf`, `vswprintf`, `vfwscanf`, `vwprintf`, `vwscanf`, and `vswscanf` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.
The `vfwscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vfwscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.7 The `vswprintf` function

Synopsis

```c
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);
```

Description

The `vswprintf` function is equivalent to `swprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vswprintf` function does not invoke the `va_end` macro.\(^{361}\)

Returns

The `vswprintf` function returns the number of wide characters written in the array, not counting the terminating null wide character, or a negative value if an encoding error occurred or if `n` or more wide characters were requested to be generated.

7.29.2.8 The `vswscanf` function

Synopsis

```c
#include <stdarg.h>
#include <wchar.h>
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
```

Description

The `vswscanf` function is equivalent to `swscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vswscanf` function does not invoke the `va_end` macro.\(^{361}\)

Returns

The `vswscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vswscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.9 The `vwprintf` function

Synopsis

```c
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t * restrict format, va_list arg);
```

Description

The `vwprintf` function is equivalent to `wprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vwprintf` function does not invoke the `va_end` macro.\(^{361}\)
Returns

The `vwprintf` function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.10 The `vwscanf` function

Synopsis

```c
#include <stdarg.h>
#include <wchar.h>
int vwscanf(const wchar_t * restrict format, va_list arg);
```

Description

The `vwscanf` function is equivalent to `wscanf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vwscanf` function does not invoke the `va_end` macro.\(^{(361)}\)

Returns

The `vwscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `vwscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.2.11 The `wprintf` function

Synopsis

```c
#include <wchar.h>
int wprintf(const wchar_t * restrict format, ...);
```

Description

The `wprintf` function is equivalent to `fwprintf` with the argument `stdout` interposed before the arguments to `wprintf`.

Returns

The `wprintf` function returns the number of wide characters transmitted, or a negative value if an output or encoding error occurred.

7.29.2.12 The `wscanf` function

Synopsis

```c
#include <wchar.h>
int wscanf(const wchar_t * restrict format, ...);
```

Description

The `wscanf` function is equivalent to `fwscanf` with the argument `stdin` interposed before the arguments to `wscanf`.

Returns

The `wscanf` function returns the value of the macro `EOF` if an input failure occurs before the first conversion (if any) has completed. Otherwise, the `wscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

7.29.3 Wide character input/output functions

7.29.3.1 The `fgetwc` function

Synopsis

```c
#include <stdio.h>
```

\(^{(366)}\) Library § 7.29.3.1
#include <wchar.h>

wint_t fgetwc(FILE *stream);

## Description

If the end-of-file indicator for the input stream pointed to by `stream` is not set and a next wide character is present, the `fgetwc` function obtains that wide character as a `wchar_t` converted to a `wint_t` and advances the associated file position indicator for the stream (if defined).

## Returns

- If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream is set and the `fgetwc` function returns `WEOF`. Otherwise, the `fgetwc` function returns the next wide character from the input stream pointed to by `stream`. If a read error occurs, the error indicator for the stream is set and the `fgetwc` function returns `WEOF`. If an encoding error occurs (including too few bytes), the error indicator for the stream is set and the value of the macro `EILSEQ` is stored in `errno` and the `fgetwc` function returns `WEOF`.\(^{362}\)

### 7.29.3.2 The `fgetws` function

#### Synopsis

```c
#include <stdio.h>
#include <wchar.h>

wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

#### Description

The `fgetws` function reads at most one less than the number of wide characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional wide characters are read after a new-line wide character (which is retained) or after end-of-file. A null wide character is written immediately after the last wide character read into the array.

#### Returns

- The `fgetws` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read or encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

### 7.29.3.3 The `fputwc` function

#### Synopsis

```c
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t c, FILE *stream);
```

#### Description

The `fputwc` function writes the wide character specified by `c` to the output stream pointed to by `stream`, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

#### Returns

- The `fputwc` function returns the wide character written. If a write error occurs, the error indicator for the stream is set and `fputwc` returns `WEOF`. If an encoding error occurs, the value of the macro `EILSEQ` is stored in `errno` and `fputwc` returns `WEOF`.\(^{362}\)

### 7.29.3.4 The `fputws` function

\(^{362}\)An end-of-file and a read error can be distinguished by use of the `feof` and `ferror` functions. Also, `errno` will be set to `EILSEQ` by input/output functions only if an encoding error occurs.
Synopsis

```c
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t * restrict s, FILE * restrict stream);
```

Description

The `fputws` function writes the wide string pointed to by `s` to the stream pointed to by `stream`. The terminating null wide character is not written.

Returns

The `fputws` function returns `EOF` if a write or encoding error occurs; otherwise, it returns a nonnegative value.

7.29.3.5 The `fwide` function

Synopsis

```c
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

Description

The `fwide` function determines the orientation of the stream pointed to by `stream`. If `mode` is greater than zero, the function first attempts to make the stream wide oriented. If `mode` is less than zero, the function first attempts to make the stream byte oriented. Otherwise, `mode` is zero and the function does not alter the orientation of the stream.

Returns

The `fwide` function returns a value greater than zero if, after the call, the stream has wide orientation, a value less than zero if the stream has byte orientation, or zero if the stream has no orientation.

7.29.3.6 The `getwc` function

Synopsis

```c
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);
```

Description

The `getwc` function is equivalent to `fgetwc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects.

Returns

The `getwc` function returns the next wide character from the input stream pointed to by `stream`, or `WEOF`.

7.29.3.7 The `getwchar` function

Synopsis

```c
#include <wchar.h>
wint_t getwchar(void);
```

Description

The `getwchar` function is equivalent to `getwc` with the argument `stdin`.

\[363\] If the orientation of the stream has already been determined, `fwide` does not change it.
Returns
3 The getwchar function returns the next wide character from the input stream pointed to by stdin, or WEOF.

7.29.3.8 The putwc function
Synopsis
1 #include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t c, FILE *stream);

Description
2 The putwc function is equivalent to fputwc, except that if it is implemented as a macro, it may evaluate stream more than once, so that argument should never be an expression with side effects.

Returns
3 The putwc function returns the wide character written, or WEOF.

7.29.3.9 The putwchar function
Synopsis
1 #include <wchar.h>

wint_t putwchar(wchar_t c);

Description
2 The putwchar function is equivalent to putwc with the second argument stdout.

Returns
3 The putwchar function returns the character written, or WEOF.

7.29.3.10 The ungetwc function
Synopsis
1 #include <stdio.h>
#include <wchar.h>

wint_t ungetwc(wint_t c, FILE *stream);

Description
2 The ungetwc function pushes the wide character specified by c back onto the input stream pointed to by stream. Pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file positioning function (fseek, fsetpos, or rewind) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged.

3 One wide character of pushback is guaranteed, even if the call to the ungetwc function follows just after a call to a formatted wide character input function (fscanf, fsetpos, or rewind) discards any pushed-back wide characters for the stream. If the ungetwc function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

4 If the value of c equals that of the macro WEOF, the operation fails and the input stream is unchanged.

5 A successful call to the ungetwc function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.\footnote{Note that a file positioning function could further modify the file position indicator after discarding any pushed-back wide characters.} For a text or binary stream, the value of its file position indicator after a successful call to the ungetwc function is unspecified until all pushed-back wide characters are read or discarded.
Returns

6 The ungetwc function returns the wide character pushed back, or WEOF if the operation fails.

7.29.4 General wide string utilities

1 The header <wchar.h> declares a number of functions useful for wide string manipulation. Various methods are used for determining the lengths of the arrays, but in all cases a wchar_t * argument points to the initial (lowest addressed) element of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

2 Where an argument declared as size_t n determines the length of the array for a function, n can have the value zero on a call to that function. Unless explicitly stated otherwise in the description of a particular function in this subclause, pointer arguments on such a call shall still have valid values, as described in 7.1.4. On such a call, a function that locates a wide character finds no occurrence, a function that compares two wide character sequences returns zero, and a function that copies wide characters copies zero wide characters.

7.29.4.1 Wide string numeric conversion functions

7.29.4.1.1 The wcstod, wcstof, and wcstold functions

Synopsis

```c
#include <wchar.h>

double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

2 The wcstod, wcstof, and wcstold functions convert the initial portion of the wide string pointed to by nptr to double, float, and long double representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the function), a subject sequence resembling a floating-point constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

3 The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte wide character in 6.4.4.2;
- a \texttt{0x} or \texttt{0X}, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary exponent part as defined in 6.4.4.2;
- \texttt{INF} or \texttt{INFINITY}, or any other wide string equivalent except for case
- \texttt{NAN} or \texttt{NAN}(\textit{n-wide-character-sequence}), or any other wide string equivalent except for case in the \texttt{NAN} part, where:

```c
n-wide-character-sequence:
  digit
  nondigit
  n-wide-character-sequence digit
  n-wide-character-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.
If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, except that the decimal-point wide character is used in place of a period, and that if neither an exponent part nor a decimal-point wide character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated.\footnote{It is unspecified whether a minus-signed sequence is converted to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (see F.5); the two methods could yield different results if rounding is toward positive or negative infinity. In either case, the functions honor the sign of zero if floating-point arithmetic supports signed zeros.} A wide character sequence \texttt{INF} or \texttt{INFINITY} is interpreted as an infinity, if representable in the return type, else like a floating constant that is too large for the range of the return type. A wide character sequence \texttt{NAN} or \texttt{NAN(n-wchar-sequenceopt)} is interpreted as a quiet NaN, if supported in the return type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequence is implementation-defined.\footnote{An implementation can use the n-wchar sequence to determine extra information to be represented in the NaN's significand.} A pointer to the final wide string is stored in the object pointed to by \texttt{endptr}, provided that \texttt{endptr} is not a null pointer.

If the subject sequence has the hexadecimal form and \texttt{FLT_RADIX} is a power of 2, the value resulting from the conversion is correctly rounded.

In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of \texttt{nptr} is stored in the object pointed to by \texttt{endptr}, provided that \texttt{endptr} is not a null pointer.

\textbf{Recommended practice}

If the subject sequence has the hexadecimal form, \texttt{FLT_RADIX} is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most $M$ significant digits, where $M$ is the maximum value of the $T$ \texttt{DECIMAL_DIG} macros (defined in \texttt{<float.h>}) significant digits, the result should be correctly rounded. If the subject sequence $D$ has the decimal form and more than $\texttt{DECIMAL_DIG~M}$ significant digits, consider the two bounding, adjacent decimal strings $L$ and $U$, both having $\texttt{DECIMAL_DIG~M}$ significant digits, such that the values of $L$, $D$, and $U$ satisfy $L \leq D \leq U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding $L$ and $U$ according to the current rounding direction, with the extra stipulation that the error with respect to $D$ should have a correct sign for the current rounding direction.\footnote{$M$ is sufficiently large that $L$ and $U$ will usually correctly round to the same internal floating value, but if not will correctly round to adjacent values.}

\textbf{Returns}

The functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value overflows and default rounding is in effect (7.12.1), plus or minus \texttt{HUGE_VAL}, \texttt{HUGE_VALF}, or \texttt{HUGE_VALL} is returned (according to the return type and sign of the value), and the value of the macro \texttt{ERANGE} is stored in \texttt{errno}. If the result underflows (7.12.1), the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type; whether \texttt{errno} acquires the value \texttt{ERANGE} is implementation-defined.

\subsection*{7.29.1.2 The wcstodN functions}

\textbf{Synopsis}

\begin{verbatim}
#include <wchar.h>

#ifdef __STDC_IEC_60559_DFP__
  _Decimal32 wcstod32(const wchar_t * restrict nptr, char ** restrict endptr);
  _Decimal64 wcstod64(const wchar_t * restrict nptr, char ** restrict endptr);
#endif
\end{verbatim}
Description

The `wcstodN` functions convert the initial portion of the wide string pointed to by `nptr` to decimal floating type representation. First, they decompose the input wide string into three parts: an initial, possibly empty, sequence of white-space wide characters; a subject sequence resembling a floating constant or representing an infinity or NaN; and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined in 6.4.4.2
- `INF` or `INFINITY`, ignoring case
- `NAN` or `NAN(d-wchar-sequence_opt)`, ignoring case in the `NAN` part, where:
  `d-wchar-sequence`:
  - digit
  - nondigit
  - `d-wchar-sequence digit`
  - `d-wchar-sequence nondigit`

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of 6.4.4.2, including correct rounding and determination of the coefficient `c` and the quantum exponent `q`, with the following exceptions:

- It is not a hexadecimal floating number.
- The decimal-point wide character is used in place of a period.
- If neither an exponent part nor a decimal-point wide character appears in a decimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the wide string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated (before rounding) and the sign `s` is set to −1, else `s` is set to 1. A wide character sequence `INF` or `INFINITY` is interpreted as an infinity. A wide character sequence `NAN` or `NAN(d-wchar-sequence_opt)`, is interpreted as a quiet NaN; the meaning of the d-wchar sequence is implementation-defined. A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Footnote: An implementation may use the d-wchar sequence to determine extra information to be represented in the NaN's significand.
Returns
The `wcstod` functions return the correctly rounded converted value, if any. If no conversion could be performed, the value of the triple `(1, 0, 0)` is returned. If the correct value overflows, the value of the macro `ERANGE` is stored in `errno`. If the result underflows (7.12.1), whether `errno` acquires the value `ERANGE` is implementation-defined.

7.29.4.1.3 The `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions

Synopsis
```c
#include <wchar.h>

long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

Description
The `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions convert the initial portion of the wide string pointed to by `nptr` to `long int`, `long long int`, `unsigned long int`, and `unsigned long long int` representation, respectively. First, they decompose the input string into three parts: an initial, possibly empty, sequence of white-space wide characters (as specified by the function), a subject sequence resembling an integer represented in some radix determined by the value of `base`, and a final wide string of one or more unrecognized wide characters, including the terminating null wide character of the input wide string. Then, they attempt to convert the subject sequence to an integer, and return the result.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in 6.4.4.1, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white-space wide characters, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 through 35; only letters and digits whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence contains no wide characters if the input wide string is empty or consists entirely of white-space wide characters, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of 6.4.4.1. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. The subject sequence begins with a minus sign, the value resulting from the conversion is negated (in the return type). A pointer to the final wide string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the "C" locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.
Returns
8 The `wcstol`, `wcstoll`, `wcstoul`, and `wcstoull` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, or `ULLONG_MAX` is returned (according to the return type sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

7.29.4.2 Wide string copying functions
7.29.4.2.1 The `wcscpy` function

Synopsis
1

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Description
2 The `wcscpy` function copies the wide string pointed to by `s2` (including the terminating null wide character) into the array pointed to by `s1`.

Returns
3 The `wcscpy` function returns the value of `s1`.

7.29.4.2.2 The `wcsncpy` function

Synopsis
1

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description
2 The `wcsncpy` function copies not more than `n` wide characters (those that follow a null wide character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`.

3 If the array pointed to by `s2` is a wide string that is shorter than `n` wide characters, null wide characters are appended to the copy in the array pointed to by `s1`, until `n` wide characters in all have been written.

Returns
4 The `wcsncpy` function returns the value of `s1`.

7.29.4.2.3 The `wmemcpy` function

Synopsis
1

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description
2 The `wmemcpy` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`.

Returns
3 The `wmemcpy` function returns the value of `s1`.

7.29.4.2.4 The `wmemmove` function

Synopsis
1

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Thus, if there is no null wide character in the first `n` wide characters of the array pointed to by `s2`, the result will not be null-terminated.
Description
2 The `wmemmove` function copies \( n \) wide characters from the object pointed to by `s2` to the object pointed to by `s1`. Copying takes place as if the \( n \) wide characters from the object pointed to by `s2` are first copied into a temporary array of \( n \) wide characters that does not overlap the objects pointed to by `s1` or `s2`, and then the \( n \) wide characters from the temporary array are copied into the object pointed to by `s1`.

Returns
3 The `wmemmove` function returns the value of `s1`.

7.29.4.3 Wide string concatenation functions
7.29.4.3.1 The `wcscat` function

Synopsis
1
```c
#include <wchar.h>
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
```

Description
2 The `wcscat` function appends a copy of the wide string pointed to by `s2` (including the terminating null wide character) to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`.

Returns
3 The `wcscat` function returns the value of `s1`.

7.29.4.3.2 The `wcsncat` function

Synopsis
1
```c
#include <wchar.h>
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
```

Description
2 The `wcsncat` function appends not more than \( n \) wide characters (a null wide character and those that follow it are not appended) from the array pointed to by `s2` to the end of the wide string pointed to by `s1`. The initial wide character of `s2` overwrites the null wide character at the end of `s1`. A terminating null wide character is always appended to the result.\(^{370}\)

Returns
3 The `wcsncat` function returns the value of `s1`.

7.29.4.4 Wide string comparison functions

Unless explicitly stated otherwise, the functions described in this subclause order two wide characters the same way as two integers of the underlying integer type designated by `wchar_t`.

7.29.4.4.1 The `wcscmp` function

Synopsis
1
```c
#include <wchar.h>
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Description
2 The `wcscmp` function compares the wide string pointed to by `s1` to the wide string pointed to by `s2`.

Returns
3 The `wcscmp` function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by `s1` is greater than, equal to, or less than the wide string pointed to by `s2`.

\(^{370}\) Thus, the maximum number of wide characters that can end up in the array pointed to by `s1` is `wcslen(s1)+n+1`.
7.29.4.4.2 The wcs.coll function

Synopsis

```
#include <wchar.h>
int wcs.coll(const wchar_t *s1, const wchar_t *s2);
```

Description

The wcs.coll function compares the wide string pointed to by s1 to the wide string pointed to by s2, both interpreted as appropriate to the LC_COLLATE category of the current locale.

Returns

The wcs.coll function returns an integer greater than, equal to, or less than zero, accordingly as the wide string pointed to by s1 is greater than, equal to, or less than the wide string pointed to by s2 when both are interpreted as appropriate to the current locale.

7.29.4.4.3 The wcsncmp function

Synopsis

```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The wcsncmp function compares not more than n wide characters (those that follow a null wide character are not compared) from the array pointed to by s1 to the array pointed to by s2.

Returns

The wcsncmp function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by s1 is greater than, equal to, or less than the possibly null-terminated array pointed to by s2.

7.29.4.4.4 The wcsxfrm function

Synopsis

```
#include <wchar.h>
size_t wcsxfrm(wchar_t *restrict s1, const wchar_t *restrict s2, size_t n);
```

Description

The wcsxfrm function transforms the wide string pointed to by s2 and places the resulting wide string into the array pointed to by s1. The transformation is such that if the wcsncmp function is applied to two transformed wide strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the wcs.coll function applied to the same two original wide strings. No more than n wide characters are placed into the resulting array pointed to by s1, including the terminating null wide character. If n is zero, s1 is permitted to be a null pointer.

Returns

The wcsxfrm function returns the length of the transformed wide string (not including the terminating null wide character). If the value returned is n or greater, the contents of the array pointed to by s1 are indeterminate.

Example

The value of the following expression is the length of the array needed to hold the transformation of the wide string pointed to by s:

```
1 + wcsxfrm(NULL, s, 0)
```

7.29.4.4.5 The wmemcmp function

Synopsis

```
#include <wchar.h>
```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);

Description
The `wmemcmp` function compares the first `n` wide characters of the object pointed to by `s1` to the first `n` wide characters of the object pointed to by `s2`.

Returns
The `wmemcmp` function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

7.29.4.5 Wide string search functions
7.29.4.5.1 The `wcschr` function
Synopsis
```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description
The `wcschr` function locates the first occurrence of `c` in the wide string pointed to by `s`. The terminating null wide character is considered to be part of the wide string.

Returns
The `wcschr` function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the wide string.

7.29.4.5.2 The `wcscspn` function
Synopsis
```
#include <wchar.h>
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description
The `wcscspn` function computes the length of the maximum initial segment of the wide string pointed to by `s1` which consists entirely of wide characters not from the wide string pointed to by `s2`.

Returns
The `wcscspn` function returns the length of the segment.
7.29.4.5.3 The wcspbrk function

Synopsis

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

Description

The `wcspbrk` function locates the first occurrence in the wide string pointed to by `s1` of any wide character from the wide string pointed to by `s2`.

Returns

The `wcspbrk` function returns a pointer to the wide character in `s1`, or a null pointer if no wide character from `s2` occurs in `s1`.

7.29.4.5.4 The wcsrchr function

Synopsis

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

The `wcsrchr` function locates the last occurrence of `c` in the wide string pointed to by `s`. The terminating null wide character is considered to be part of the wide string.

Returns

The `wcsrchr` function returns a pointer to the wide character, or a null pointer if `c` does not occur in the wide string.

7.29.4.5.5 The wcsspn function

Synopsis

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Description

The `wcsspn` function computes the length of the maximum initial segment of the wide string pointed to by `s1` which consists entirely of wide characters from the wide string pointed to by `s2`.

Returns

The `wcsspn` function returns the length of the segment.

7.29.4.5.6 The wcsstr function

Synopsis

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Description

The `wcsstr` function locates the first occurrence in the wide string pointed to by `s1` of the sequence of wide characters (excluding the terminating null wide character) in the wide string pointed to by `s2`.

Returns

The `wcsstr` function returns a pointer to the located wide string, or a null pointer if the wide string is not found. If `s2` points to a wide string with zero length, the function returns `s1`. 
7.29.4.5.7 The \texttt{wcstok} function

Synopsis

\begin{verbatim}
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t ** restrict ptr);
\end{verbatim}

Description

A sequence of calls to the \texttt{wcstok} function breaks the wide string pointed to by \texttt{s1} into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by \texttt{s2}. The third argument points to a caller-provided \texttt{wchar_t} pointer into which the \texttt{wcstok} function stores information necessary for it to continue scanning the same wide string.

The first call in a sequence has a non-null first argument and stores an initial value in the object pointed to by \texttt{ptr}. Subsequent calls in the sequence have a null first argument and the object pointed to by \texttt{ptr} is required to have the value stored by the previous call in the sequence, which is then updated. The separator wide string pointed to by \texttt{s2} may be different from call to call.

The first call in the sequence searches the wide string pointed to by \texttt{s1} for the first wide character that is not contained in the current separator wide string pointed to by \texttt{s2}. If no such wide character is found, then there are no tokens in the wide string pointed to by \texttt{s1} and the \texttt{wcstok} function returns a null pointer. If such a wide character is found, it is the start of the first token.

The \texttt{wcstok} function then searches from there for a wide character that is contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by \texttt{s1}, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

In all cases, the \texttt{wcstok} function stores sufficient information in the pointer pointed to by \texttt{ptr} so that subsequent calls, with a null pointer for \texttt{s1} and the unmodified pointer value for \texttt{ptr}, shall start searching just past the element overwritten by a null wide character (if any).

Returns

The \texttt{wcstok} function returns a pointer to the first wide character of a token, or a null pointer if there is no token.

\begin{verbatim}
#include <wchar.h>
static wchar_t str1[] = L"?a??b,,,,#c"; static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2; t = wcstok(str1, L"?", &ptr1);       // t points to the token L"a" t = wcstok(str1, L"\t", &ptr1);       // t points to the token L"?b" t = wcstok(str2, L"\t", &ptr2);       // t is a null pointer t = wcstok(NULL, L"#", &ptr1);       // t points to the token L"c" t = wcstok(NULL, L"?", &ptr1);       // t is a null pointer
\end{verbatim}

7.29.4.5.8 The \texttt{wmemchr} function

Synopsis

\begin{verbatim}
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
\end{verbatim}

Description

The \texttt{wmemchr} function locates the first occurrence of \texttt{c} in the initial \texttt{n} wide characters of the object pointed to by \texttt{s}.

\section*{Library 379}
Returns
3 The \texttt{wmemchr} function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

7.29.4.6 Miscellaneous functions
7.29.4.6.1 The \texttt{wcslen} function
Synopsis
1
\begin{verbatim}
#include <wchar.h>
size_t wcslen(const wchar_t *s);
\end{verbatim}

Description
2 The \texttt{wcslen} function computes the length of the wide string pointed to by \texttt{s}.

Returns
3 The \texttt{wcslen} function returns the number of wide characters that precede the terminating null wide character.

7.29.4.6.2 The \texttt{wmemset} function
Synopsis
1
\begin{verbatim}
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
\end{verbatim}

Description
2 The \texttt{wmemset} function copies the value of \texttt{c} into each of the first \texttt{n} wide characters of the object pointed to by \texttt{s}.

Returns
3 The \texttt{wmemset} function returns the value of \texttt{s}.

7.29.5 Wide character time conversion functions
7.29.5.1 The \texttt{wcsftime} function
Synopsis
1
\begin{verbatim}
#include <time.h>
#include <wchar.h>
size_t wcsftime(wchar_t * restrict s, size_t maxsize, const wchar_t * restrict format, const struct tm * restrict timeptr);
\end{verbatim}

Description
2 The \texttt{wcsftime} function is equivalent to the \texttt{strftime} function, except that:

\begin{itemize}
\item The argument \texttt{s} points to the initial element of an array of wide characters into which the generated output is to be placed.
\item The argument \texttt{maxsize} indicates the limiting number of wide characters.
\item The argument \texttt{format} is a wide string and the conversion specifiers are replaced by corresponding sequences of wide characters.
\item The return value indicates the number of wide characters.
\end{itemize}

Returns
3 If the total number of resulting wide characters including the terminating null wide character is not more than \texttt{maxsize}, the \texttt{wcsftime} function returns the number of wide characters placed into the array pointed to by \texttt{s} not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.
7.29.6  Extended multibyte/wide character conversion utilities

The header `<wchar.h>` declares an extended set of functions useful for conversion between multibyte characters and wide characters.

Most of the following functions — those that are listed as “restartable”, 7.29.6.3 and 7.29.6.4 — take as a last argument a pointer to an object of type `mbstate_t` that is used to describe the current conversion state from a particular multibyte character sequence to a wide character sequence (or the reverse) under the rules of a particular setting for the `LC_CTYPE` category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new multibyte character in the initial shift state. A zero-valued `mbstate_t` object is (at least) one way to describe an initial conversion state. A zero-valued `mbstate_t` object can be used to initiate conversion involving any multibyte character sequence, in any `LC_CTYPE` category setting. If an `mbstate_t` object has been altered by any of the functions described in this subclause, and is then used with a different multibyte character sequence, or in the other conversion direction, or with a different `LC_CTYPE` category setting than on earlier function calls, the behavior is undefined.  

On entry, each function takes the described conversion state (either internal or pointed to by an argument) as current. The conversion state described by the referenced object is altered as needed to track the shift state, and the position within a multibyte character, for the associated multibyte character sequence.

7.29.6.1  Single-byte/wide character conversion functions

7.29.6.1.1  The `btowc` function

Synopsis

```c
#include <wchar.h>

wint_t btowc(int c);
```

Description

The `btowc` function determines whether `c` constitutes a valid single-byte character in the initial shift state.

Returns

The `btowc` function returns `WEOF` if `c` has the value `EOF` or if `(unsigned char)c` does not constitute a valid single-byte character in the initial shift state. Otherwise, it returns the wide character representation of that character.

7.29.6.1.2  The `wctob` function

Synopsis

```c
#include <wchar.h>

int wctob(wint_t c);
```

Description

The `wctob` function determines whether `c` corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.

Returns

The `wctob` function returns `EOF` if `c` does not correspond to a multibyte character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character as an `unsigned char` converted to an `int`.

7.29.6.2  Conversion state functions

7.29.6.2.1  The `mbsinit` function

Thus, a particular `mbstate_t` object can be used, for example, with both the `mbtowc` and `mbsrtowcs` functions as long as they are used to step sequentially through the same multibyte character string.

§ 7.29.6.2.1  Library
Synopsis

```c
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

Description

If `ps` is not a null pointer, the `mbsinit` function determines whether the referenced `mbstate_t` object describes an initial conversion state.

Returns

The `mbsinit` function returns nonzero if `ps` is a null pointer or if the referenced object describes an initial conversion state; otherwise, it returns zero.

7.29.6.3 Restartable multibyte/wide character conversion functions

These functions differ from the corresponding multibyte character functions of 7.22.7 (`mblen`, `mbtowc`, and `wctomb`) in that they have an extra parameter, `ps`, of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a null pointer, each function uses its own internal `mbstate_t` object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for `ps`.

Also unlike their corresponding functions, the return value does not represent whether the encoding is state-dependent.

7.29.6.3.1 The `mbrlen` function

Synopsis

```c
#include <wchar.h>
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

The `mbrlen` function is equivalent to the call:

```c
mbrtowc(NULL, s, n, ps != NULL ? ps: &internal)
```

where `internal` is the `mbstate_t` object for the `mbrlen` function, except that the expression designated by `ps` is evaluated only once.

Returns

The `mbrlen` function returns a value between zero and `n`, inclusive, \((\text{size}_t)(-2)\), or \((\text{size}_t)(-1)\).

Forward references: the `mbtowc` function (7.29.6.3.2).

7.29.6.3.2 The `mbtowc` function

Synopsis

```c
#include <wchar.h>
size_t mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

If `s` is a null pointer, the `mbtowc` function is equivalent to the call:

```c
mbtowc(NULL, ",", 1, ps)
```

In this case, the values of the parameters `pwc` and `n` are ignored.
If \( s \) is not a null pointer, the \texttt{mbrtowc} function inspects at most \( n \) bytes beginning with the byte pointed to by \( s \) to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the value of the corresponding wide character and then, if \( \text{pwc} \) is not a null pointer, stores that value in the object pointed to by \( \text{pwc} \). If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

Returns

The \texttt{mbrtowc} function returns the first of the following that applies (given the current conversion state):

- \( 0 \) if the next \( n \) or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).
- \( \textbf{between } 1 \textbf{ and } n \textbf{ inclusive} \) if the next \( n \) or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.
- \( \textbf{(size\_t)}(-2) \) if the next \( n \) bytes contribute to an incomplete (but potentially valid) multibyte character, and all \( n \) bytes have been processed (no value is stored).
- \( \textbf{(size\_t)}(-1) \) if an encoding error occurs, in which case the next \( n \) or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored); the value of the macro \texttt{EILSEQ} is stored in \texttt{errno}, and the conversion state is unspecified.

7.29.6.3.3 The \texttt{wcrtomb} function

Synopsis

```c
#include <wchar.h>
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

Description

If \( s \) is a null pointer, the \texttt{wcrtomb} function is equivalent to the call

```c
wcrtomb(buf, L'\0', ps)
```

where \texttt{buf} is an internal buffer.

If \( s \) is not a null pointer, the \texttt{wcrtomb} function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by \( \text{wc} \) (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by \( s \). At most \texttt{MB\_CUR\_MAX} bytes are stored. If \( \text{wc} \) is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

Returns

The \texttt{wcrtomb} function returns the number of bytes stored in the array object (including any shift sequences). When \( \text{wc} \) is not a valid wide character, an encoding error occurs: the function stores the value of the macro \texttt{EILSEQ} in \texttt{errno} and returns \( \textbf{(size\_t)}(-1) \); the conversion state is unspecified.

7.29.6.4 Restartable multibyte/wide string conversion functions

These functions differ from the corresponding multibyte string functions of 7.22.8 (\texttt{mbstowcs} and \texttt{wcstombs}) in that they have an extra parameter, \texttt{ps}, of type pointer to \texttt{mbstate\_t} that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If \texttt{ps} is a null pointer, each function uses its own internal \texttt{mbstate\_t} object instead, which is initialized at program startup to the initial conversion state; the functions are not required to avoid redundant shift sequences (for implementations with state-dependent encodings).
data races with other calls to the same function in this case. The implementation behaves as if no library function calls these functions with a null pointer for \( ps \).

Also unlike their corresponding functions, the conversion source parameter, \( src \), has a pointer-to-pointer type. When the function is storing the results of conversions (that is, when \( dst \) is not a null pointer), the pointer object pointed to by this parameter is updated to reflect the amount of the source processed by that invocation.

### 7.29.6.4.1 The `mbsrtowcs` function

**Synopsis**

```c
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);
```

**Description**

The `mbsrtowcs` function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by \( ps \), from the array indirectly pointed to by \( src \) into a sequence of corresponding wide characters. If \( dst \) is not a null pointer, the converted characters are stored into the array pointed to by \( dst \). Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if \( dst \) is not a null pointer) when \( len \) wide characters have been stored into the array pointed to by \( dst \). Each conversion takes place as if by a call to the `mbrtowc` function.

If \( dst \) is not a null pointer, the pointer object pointed to by \( src \) is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if \( dst \) is not a null pointer, the resulting state described is the initial conversion state.

**Returns**

If the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the `mbsrtowcs` function stores the value of the macro `EILSEQ` in `errno` and returns \((size_t)(-1)\); the conversion state is unspecified. Otherwise, it returns the number of multibyte characters successfully converted, not including the terminating null character (if any).

### 7.29.6.4.2 The `wcsrtombs` function

**Synopsis**

```c
#include <wchar.h>
size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src, size_t len, mbstate_t * restrict ps);
```

**Description**

The `wcsrtombs` function converts a sequence of wide characters from the array indirectly pointed to by \( src \) into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by \( ps \). If \( dst \) is not a null pointer, the converted characters are then stored into the array pointed to by \( dst \). Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases: when a wide character is reached that does not correspond to a valid multibyte character, or (if \( dst \) is not a null pointer) when the next multibyte character would exceed the limit of \( len \) total bytes to be stored into the array pointed to by \( dst \). Each conversion takes place as if by a call to the `wcrtomb` function.

If \( dst \) is not a null pointer, the pointer object pointed to by \( src \) is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the

---

\[373\] Thus, the value of \( len \) is ignored if \( dst \) is a null pointer.

\[374\] If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte.
last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

Returns

4 If conversion stops because a wide character is reached that does not correspond to a valid multibyte character, an encoding error occurs: the wcsrtombs function stores the value of the macro EILSEQ in errno and returns (size_t) (−1); the conversion state is unspecified. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).
7.30 Wide character classification and mapping utilities <wctype.h>

7.30.1 Introduction

The header <wctype.h> defines one macro, and declares three data types and many functions.375) The types declared are `wint_t` described in 7.29.1;

```c
    wctrans_t
```

which is a scalar type that can hold values which represent locale-specific character mappings; and

```c
    wctype_t
```

which is a scalar type that can hold values which represent locale-specific character classifications.

The macro defined is `WEOF` (described in 7.29.1).

The functions declared are grouped as follows:

- Functions that provide wide character classification;
- Extensible functions that provide wide character classification;
- Functions that provide wide character case mapping;
- Extensible functions that provide wide character mapping.

For all functions described in this subclause that accept an argument of type `wint_t`, the value shall be representable as a `wchar_t` or shall equal the value of the macro `WEOF`. If this argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the `LC_CTYPE` category of the current locale.

7.30.2 Wide character classification utilities

The header <wctype.h> declares several functions useful for classifying wide characters.

The term `printing wide character` refers to a member of a locale-specific set of wide characters, each of which occupies at least one printing position on a display device. The term `control wide character` refers to a member of a locale-specific set of wide characters that are not printing wide characters.

7.30.2.1 Wide character classification functions

The functions in this subclause return nonzero (true) if and only if the value of the argument `wc` conforms to that in the description of the function.

Each of the following functions returns true for each wide character that corresponds (as if by a call to the `wctob` function) to a single-byte character for which the corresponding character classification function from 7.4.1 returns true, except that the `iswgraph` and `iswpunct` functions may differ with respect to wide characters other than `L''` that are both printing and white-space wide characters.376)

Forward references: the `wctob` function (7.29.6.1.2).

---

375) See “future library directions” (7.31.19).
376) For example, if the expression `isalpha(wctob(wc))` evaluates to true, then the call `iswalpha(wc)` also returns true. But, if the expression `isgraph(wctob(wc))` evaluates to true (which cannot occur for `wc == L''` of course), then either `iswgraph(wc)` or `iswprint(wc)&& iswspace(wc)` is true, but not both.
7.30.2.1.1 The iswalnum function

Synopsis

```c
#include <wctype.h>
int iswalnum(wint_t wc);
```

Description

The `iswalnum` function tests for any wide character for which `iswalpha` or `iswdigit` is true.

7.30.2.1.2 The iswalpha function

Synopsis

```c
#include <wctype.h>
int iswalpha(wint_t wc);
```

Description

The `iswalpha` function tests for any wide character for which `iswupper` or `iswlower` is true, or any wide character that is one of a locale-specific set of alphabetic wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.\(^{377}\)

7.30.2.1.3 The iswblank function

Synopsis

```c
#include <wctype.h>
int iswblank(wint_t wc);
```

Description

The `iswblank` function tests for any wide character that is a standard blank wide character or is one of a locale-specific set of wide characters for which `iswspace` is true and that is used to separate words within a line of text. The standard blank wide characters are the following: space (L’ ’), and horizontal tab (L’\t’). In the "C" locale, `iswblank` returns true only for the standard blank characters.

7.30.2.1.4 The iswcntrl function

Synopsis

```c
#include <wctype.h>
int iswcntrl(wint_t wc);
```

Description

The `iswcntrl` function tests for any control wide character.

7.30.2.1.5 The iswdigit function

Synopsis

```c
#include <wctype.h>
int iswdigit(wint_t wc);
```

Description

The `iswdigit` function tests for any wide character that corresponds to a decimal-digit character (as defined in 5.2.1).

\(^{377}\)The functions `iswlower` and `iswupper` test true or false separately for each of these additional wide characters; all four combinations are possible.
7.30.2.1.6 The iswgraph function

Synopsis

```
#include <wctype.h>
int iswgraph(wint_t wc);
```

Description

The `iswgraph` function tests for any wide character for which `iswprint` is true and `iswspace` is false.\(^\text{378}\)

7.30.2.1.7 The iswlower function

Synopsis

```
#include <wctype.h>
int iswlower(wint_t wc);
```

Description

The `iswlower` function tests for any wide character that corresponds to a lowercase letter or is one of a locale-specific set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.

7.30.2.1.8 The iswprint function

Synopsis

```
#include <wctype.h>
int iswprint(wint_t wc);
```

Description

The `iswprint` function tests for any printing wide character.

7.30.2.1.9 The iswpunct function

Synopsis

```
#include <wctype.h>
int iswpunct(wint_t wc);
```

Description

The `iswpunct` function tests for any printing wide character that is one of a locale-specific set of punctuation wide characters for which neither `iswspace` nor `iswalnum` is true.\(^\text{378}\)

7.30.2.1.10 The iswspace function

Synopsis

```
#include <wctype.h>
int iswspace(wint_t wc);
```

Description

The `iswspace` function tests for any wide character that corresponds to a locale-specific set of white-space wide characters for which none of `iswalnum`, `iswgraph`, or `iswpunct` is true.

7.30.2.1.11 The iswupper function

Synopsis

```
#include <wctype.h>
int iswupper(wint_t wc);
```

\(^{378}\)Note that the behavior of the `iswgraph` and `iswpunct` functions can differ from their corresponding functions in 7.4.1 with respect to printing, white-space, single-byte execution characters other than ‘ ‘.
Description
2 The `iswupper` function tests for any wide character that corresponds to an uppercase letter or is one of a locale-specific set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.

7.30.2.1.12 The `iswxdigit` function

Synopsis
1
```
#include <wctype.h>
int iswxdigit(wint_t wc);
```

Description
2 The `iswxdigit` function tests for any wide character that corresponds to a hexadecimal-digit character (as defined in 6.4.4.1).

7.30.2.2 Extensible wide character classification functions
1 The functions `wctype` and `iswctype` provide extensible wide character classification as well as testing equivalent to that performed by the functions described in the previous subclause (7.30.2.1).

7.30.2.2.1 The `iswctype` function

Synopsis
1
```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t desc);
```

Description
2 The `iswctype` function determines whether the wide character `wc` has the property described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctype` that returned the value `desc`.

3 Each of the following expressions has a truth-value equivalent to the call to the wide character classification function (7.30.2.1) in the comment that follows the expression:

```
iswctype(wc, wctype("alnum")) // iswalnum(wc)
iswctype(wc, wctype("alpha")) // iswalpha(wc)
iswctype(wc, wctype("blank")) // iswblank(wc)
iswctype(wc, wctype("cntrl")) // iswcntrl(wc)
iswctype(wc, wctype("digit")) // iswdigit(wc)
iswctype(wc, wctype("graph")) // iswgraph(wc)
iswctype(wc, wctype("lower")) // iswlower(wc)
iswctype(wc, wctype("print")) // iswprint(wc)
iswctype(wc, wctype("punct")) // iswpunct(wc)
iswctype(wc, wctype("space")) // iswspace(wc)
iswctype(wc, wctype("upper")) // iswupper(wc)
iswctype(wc, wctype("xdigit")) // iswxdigit(wc)
```

Returns
4 The `iswctype` function returns nonzero (true) if and only if the value of the wide character `wc` has the property described by `desc`. If `desc` is zero, the `iswctype` function returns zero (false).

Forward references: the `wctype` function (7.30.2.2.2).

7.30.2.2.2 The `wctype` function

Synopsis
1
```
#include <wctype.h>
wctype_t wctype(const char *property);
```
### Description

2. The `wctype` function constructs a value with type `wctype_t` that describes a class of wide characters identified by the string argument `property`.

3. The strings listed in the description of the `iswctype` function shall be valid in all locales as `property` arguments to the `wctype` function.

### Returns

4. If `property` identifies a valid class of wide characters according to the `LC_CTYPE` category of the current locale, the `wctype` function returns a nonzero value that is valid as the second argument to the `iswctype` function; otherwise, it returns zero.

#### 7.30.3 Wide character case mapping utilities

1. The header `<wctype.h>` declares several functions useful for mapping wide characters.

#### 7.30.3.1 Wide character case mapping functions

**7.30.3.1.1 The `towlower` function**

*Synopsis*

```c
#include <wctype.h>
wint_t towlower(wint_t wc);
```

*Description*

2. The `towlower` function converts an uppercase letter to a corresponding lowercase letter.

*Returns*

3. If the argument is a wide character for which `iswupper` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswlower` is true, the `towlower` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

**7.30.3.1.2 The `towupper` function**

*Synopsis*

```c
#include <wctype.h>
wint_t towupper(wint_t wc);
```

*Description*

2. The `towupper` function converts a lowercase letter to a corresponding uppercase letter.

*Returns*

3. If the argument is a wide character for which `iswlower` is true and there are one or more corresponding wide characters, as specified by the current locale, for which `iswupper` is true, the `towupper` function returns one of the corresponding wide characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

#### 7.30.3.2 Extensible wide character case mapping functions

1. The functions `wctrans` and `towctrans` provide extensible wide character mapping as well as case mapping equivalent to that performed by the functions described in the previous subclause (7.30.3.1).

**7.30.3.2.1 The `towctrans` function**

*Synopsis*

```c
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```
Description
2 The `towctrans` function maps the wide character `wc` using the mapping described by `desc`. The current setting of the `LC_CTYPE` category shall be the same as during the call to `wctrans` that returned the value `desc`.
3 Each of the following expressions behaves the same as the call to the wide character case mapping function (7.30.3.1) in the comment that follows the expression:
   ```c
   towctrans(wc, wctrans("tolower")) // towlower(wc)
   towctrans(wc, wctrans("toupper")) // towupper(wc)
   ```

Returns
4 The `towctrans` function returns the mapped value of `wc` using the mapping described by `desc`. If `desc` is zero, the `towctrans` function returns the value of `wc`.

7.30.3.2.2 The `wctrans` function
Synopsis
1 ```c
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

Description
2 The `wctrans` function constructs a value with type `wctrans_t` that describes a mapping between wide characters identified by the string argument `property`.
3 The strings listed in the description of the `towctrans` function shall be valid in all locales as `property` arguments to the `wctrans` function.

Returns
4 If `property` identifies a valid mapping of wide characters according to the `LC_CTYPE` category of the current locale, the `wctrans` function returns a nonzero value that is valid as the second argument to the `towctrans` function; otherwise, it returns zero.
7.31 Future library directions

The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

7.31.1 Complex arithmetic <complex.h>

The function names

```c
const cexp10m1 clog10 crootn
cexp10 clog1p crsqrtd
ccompoundn cexp2 clog2 clgamma
ccospi cexp2m1 clog2p1 csinpi
ccospil cexpml clogmaclogpl ctgamma
cerfc cexpml clgammaclogpl cpown
ccos clog10pl cpowr
```

and the same names suffixed with `f` or `l` may be added to the declarations in the `<complex.h>` header.

7.31.2 Character handling <ctype.h>

Function names that begin with either `is` or `to`, and a lowercase letter may be added to the declarations in the `<ctype.h>` header.

7.31.3 Errors <errno.h>

Macros that begin with `E` and a digit or `E` and an uppercase letter may be added to the macros defined in the `<errno.h>` header.

7.31.4 Floating-point environment <fenv.h>

Macros that begin with `FE` and an uppercase letter may be added to the macros defined in the `<fenv.h>` header.

7.31.5 Characteristics of floating types <float.h>

Macros that begin with `DBL㎐ DEC32センチ`, `DEC64センチ`, `DEC128センチ`, `DEC∞`, `FLT∞`, or `LDBL∞` and an uppercase letter may be added to the macros defined in the `<fenv.h>` header.

7.31.6 Format conversion of integer types <inttypes.h>

Macros that begin with either `PRI` or `SCN`, and either a lowercase letter or `X` may be added to the macros defined in the `<inttypes.h>` header.

2 Function names that begin with `str`, or `wcs` and a lowercase letter may be added to the declarations in the `<inttypes.h>` header.

7.31.7 Localization <locale.h>

Macros that begin with `LC` and an uppercase letter may be added to the macros defined in the `<locale.h>` header.

7.31.8 Mathematics <math.h>

Macros that begin with `FP∞` or `MATH∞` and an uppercase letter may be added to the macros defined in the `<math.h>` header.

2 Use of the `DECIMAL_DIG` macro is an obsolescent feature. A similar type-specific macro, such as `LDBL_DECIMAL_DIG`, can be used instead.

3 Function names that begin with `is` and a lowercase letter may be added to the declarations in the `<math.h>` header.

4 The function names
and the same names suffixed with _f, _l, _d32, _d64, or _d128 may be added to the <math.h> header. The cr prefix is intended to indicate a correctly rounded version of the function.

7.31.9 Signal handling <signal.h>

Macros that begin with either SIG and an uppercase letter or SIG_ and an uppercase letter may be added to the macros defined in the <signal.h> header.

7.31.10 Atomics <stdatomic.h>

Macros that begin with ATOMIC_ and an uppercase letter may be added to the macros defined in the <stdatomic.h> header. Typedef names that begin with either atomic_ or memory_, and a lowercase letter may be added to the declarations in the <stdatomic.h> header. Enumeration constants that begin with memory_order_ and a lowercase letter may be added to the definition of the memory_order type in the <stdatomic.h> header. Function names that begin with atomic_ and a lowercase letter may be added to the declarations in the <stdatomic.h> header.

The macro ATOMIC_VAR_INIT is an obsolescent feature.

7.31.11 Boolean type and values <stdbool.h>

The ability to undefine and perhaps then redefine the macros bool, true, and false is an obsolescent feature.

7.31.12 Integer types <stdint.h>

Typedef names beginning with int or uint and ending with _t may be added to the types defined in the <stdint.h> header. Macro names beginning with INT or UINT and ending with _MAX, _MIN, _WIDTH, or _C may be added to the macros defined in the <stdint.h> header.

7.31.13 Input/output <stdio.h>

Lowercase letters may be added to the conversion specifiers and length modifiers in fprintf and fscanf. Other characters may be used in extensions.

The use of ungetc on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.31.14 General utilities <stdlib.h>

Function names that begin with str or wcs and a lowercase letter may be added to the declarations in the <stdlib.h> header.

Invoking realloc with a size argument equal to zero is an obsolescent feature.

7.31.15 String handling <string.h>

Function names that begin with str, mem, or wcs and a lowercase letter may be added to the declarations in the <string.h> header.

7.31.16 Date and time <time.h>

Macros beginning with TIME_ and an uppercase letter may be added to the macros in the <time.h> header.

§ 7.31.16 Library 393
7.31.17  Threads <threads.h>
1  Function names, type names, and enumeration constants that begin with either cnd, mtx, thrd, or tss, and a lowercase letter may be added to the declarations in the <threads.h> header.

7.31.18  Extended multibyte and wide character utilities <wchar.h>
1  Function names that begin with wcs and a lowercase letter may be added to the declarations in the <wchar.h> header.
2  Lowercase letters may be added to the conversion specifiers and length modifiers in fprintf and fscanf. Other characters may be used in extensions.

7.31.19  Wide character classification and mapping utilities <wctype.h>
1  Function names that begin with is or to and a lowercase letter may be added to the declarations in the <wctype.h> header.
Annex A
(informative)
Language syntax summary

NOTE The notation is described in 6.1.

A.1 Lexical grammar
A.1.1 Lexical elements

(6.4) token:
    keyword
    identifier
    constant
    string-literal
    punctuator

(6.4) preprocessing-token:
    header-name
    identifier
    pp-number
    character-constant
    string-literal
    punctuator
    each non-white-space character that cannot be one of the above

A.1.2 Keywords

(6.4.1) keyword: one of

auto-    signed-    case    static
break-    sizeof-    char    struct
case-     static-    const    switch
cchar-    struct-    continue    typedef
cconst-   switch-    decltype
ccontinue-    typedef-    do    unsigned
cdefault-  union-    double    void
do-       unsigned-    else    volatile
ddouble-   void-     enum    while
derelse-   volatile-    extern    _Alignas
denum-     while-     float    _Alignof
dextern-   _Alignas-    for    _Atomic
dfloat-    _Alignof-    goto    _Bool
dfor-      _Atomic-    if    _Complex
dgoto-     _Bool-     inline    _Decimal128
dif-       _Complex-    int    _Decimal32
dinline-   _Generic-    long    _Decimal64
dint-      _Imaginary-    register    _Generic
dlong-     _Noreturn-    restrict    _Imaginary
dregister-    _Static_assert-    return    _Noreturn
drestrict-    _Thread_local-    short    _Static_assert
dreturn-    auto    signed    _Thread_local
dshort-    break    sizeof

A.1.3 Identifiers

§ A.1.3 Language syntax summary 395
(6.4.2.1) **identifier**: 
  identifier-nondigit 
  identifier identifier-nondigit 
  identifier digit

(6.4.2.1) **identifier-nondigit**: 
  nondigit 
  universal-character-name 
  other implementation-defined characters

(6.4.2.1) **nondigit**: one of 
  \[a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l\ m\ n\ o\ p\ q\ r\ s\ t\ u\ v\ w\ x\ y\ z\ \]
  \[A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q\ R\ S\ T\ U\ V\ W\ X\ Y\ Z\ \]

(6.4.2.1) **digit**: one of 
  \[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ \]

---

A.1.4 **Universal character names**

(6.4.3) **universal-character-name**: 
  \[\u \ hex-quad\]
  \[\U \ hex-quad \ hex-quad\]

(6.4.3) **hex-quad**: 
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

---

A.1.5 **Constants**

(6.4.4) **constant**: 
  integer-constant 
  floating-constant 
  enumeration-constant 
  character-constant

(6.4.4.1) **integer-constant**: 
  decimal-constant integer-suffix\_opt 
  octal-constant integer-suffix\_opt 
  hexadecimal-constant integer-suffix\_opt

(6.4.4.1) **decimal-constant**: 
  nonzero-digit 
  decimal-constant digit

(6.4.4.1) **octal-constant**: 
  0 
  octal-constant octal-digit

(6.4.4.1) **hexadecimal-constant**: 
  hexadecimal-prefix hexadecimal-digit 
  hexadecimal-constant hexadecimal-digit

(6.4.4.1) **hexadecimal-prefix**: one of 
  \[-0x\ -0x\ 0x\ \]
(6.4.4.1) **nonzero-digit**: one of
\[1 2 3 4 5 6 7 8 9\]

(6.4.4.1) **octal-digit**: one of
\[0 1 2 3 4 5 6 7\]

(6.4.4.1) **hexadecimal-digit**: one of
\[0 1 2 3 4 5 6 7 8 9 \ a b c d e f \ A B C D E F\]

(6.4.4.1) integer-suffix:
- unsigned-suffix long-suffix opt
- unsigned-suffix long-long-suffix
- long-suffix unsigned-suffix opt
- long-long-suffix unsigned-suffix opt

(6.4.4.1) unsigned-suffix: one of
\[u \ U\]

(6.4.4.1) long-suffix: one of
\[l \ L\]

(6.4.4.1) long-long-suffix: one of
\[ll \ LL\]

(6.4.4.2) floating-constant:
- decimal-floating-constant
- hexadecimal-floating-constant

(6.4.4.2) decimal-floating-constant:
- fractional-constant exponent-part opt floating-suffix opt
digit-sequence exponent-part floating-suffix opt
digit-sequence exponent-part floating-suffix opt
digit-sequence exponent-part floating-suffix opt

(6.4.4.2) hexadecimal-floating-constant:
- hexadecimal-prefix hexadecimal-fractional-constant
- binary-exponent-part floating-suffix opt
- hexadecimal-prefix hexadecimal-digit-sequence
- binary-exponent-part floating-suffix opt

(6.4.4.2) fractional-constant:
- digit-sequence opt . digit-sequence
digit-sequence .

(6.4.4.2) exponent-part:
- e sign opt digit-sequence
- E sign opt digit-sequence

(6.4.4.2) sign: one of
\[+ -\]

(6.4.4.2) digit-sequence:
- digit
digit-sequence digit
digit-sequence
digit-sequence

(6.4.4.2) hexadecimal-fractional-constant:
- hexadecimal-digit-sequence opt .
  hexadecimal-digit-sequence .
(6.4.4.2) binary-exponent-part:
  \( p \) sign\(_{\text{opt}} \) digit-sequence
  \( P \) sign\(_{\text{opt}} \) digit-sequence

(6.4.4.2) hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence hexadecimal-digit

(6.4.4.2) floating-suffix: one of
  f l F L d f d d d l D F D D DL

(6.4.4.3) enumeration-constant:
  identifier

(6.4.4.4) character-constant:
  \('\) c-char-sequence \('\)
  encoding-prefix\(_{\text{opt}}\) c-char-sequence \('\)

(6.4.4.4) encoding-prefix:
  \('L\)' c-char-sequence u8
  \('U\)' c-char-sequence u
  \('U\)' c-char-sequence U

(6.4.4.4) c-char-sequence:
  c-char
  c-char-sequence c-char

(6.4.4.4) c-char:
  any member of the source character set except
  the single-quote ', backslash \, or new-line character

(6.4.4.4) escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence
  universal-character-name

(6.4.4.4) simple-escape-sequence: one of
  \('\) \"\) \?\) \\\n  \(\) \(\) \(\) \(\) \(\) \(\) \(\) \(\) \(\)

(6.4.4.4) octal-escape-sequence:
  \(\) octal-digit
  \(\) octal-digit octal-digit
  \(\) octal-digit octal-digit octal-digit

(6.4.4.4) hexadecimal-escape-sequence:
  \(x\) hexadecimal-digit
  hexadecimal-escape-sequence hexadecimal-digit

A.1.6 String literals

(6.4.5) string-literal:
  encoding-prefix\(_{\text{opt}}\)" c-char-sequence\(_{\text{opt}}\)" encoding-prefix\(_{\text{opt}}\)
  "u8"
  "u"
  "U"
  "L"
  "" s-char-sequence\(_{\text{opt}}\)"
(6.4.5) s-char-sequence:
   s-char
   s-char-sequence s-char

(6.4.5) s-char:
   any member of the source character set except
   the double-quote "", backslash \, or new-line character
   escape-sequence

A.1.7 Punctuators

(6.4.6) punctuator: one of
   [ { ( ) [ ] } ]
   ++ -- & * + - ~ !
   / % << >> < > <= >= == != ^ |
   && ||
   ::= :::: ::== :::::

A.1.8 Header names

(6.4.7) header-name:
   < h-char-sequence >
   " q-char-sequence "

(6.4.7) h-char-sequence:
   h-char
   h-char-sequence h-char

(6.4.7) h-char:
   any member of the source character set except
   the new-line character and >

(6.4.7) q-char-sequence:
   q-char
   q-char-sequence q-char

(6.4.7) q-char:
   any member of the source character set except
   the new-line character and "

A.1.9 Preprocessing numbers
(6.4.8) pp-number:

digit
  \_\_ digit
pp-number digit
pp-number identifier-nondigit
pp-number e sign
pp-number E sign
pp-number p sign
pp-number P sign
pp-number \_\_\_ sign

A.2 Phrase structure grammar
A.2.1 Expressions

(6.5.1) primary-expression:
  identifier
  constant
  string-literal
  ( expression )
  generic-selection

(6.5.1.1) generic-selection:
  _Generic ( assignment-expression , generic-assoc-list )

(6.5.1.1) generic-assoc-list:
  generic-association
  generic-assoc-list , generic-association

(6.5.1.1) generic-association:
  type-name : assignment-expression
  default: assignment-expression

(6.5.2) postfix-expression:
  primary-expression
  postfix-expression [ expression ]
  postfix-expression ( argument-expression-list_opt )
  postfix-expression . identifier
  postfix-expression -> identifier
  postfix-expression ++
  postfix-expression \_\_\_\_
  ( type-name ) { initializer-list }
  ( type-name ) { initializer-list , }

(6.5.2) argument-expression-list:
  assignment-expression
  argument-expression-list , assignment-expression

(6.5.3) unary-expression:
  postfix-expression
  ++ unary-expression
  \_\_\_\_ unary-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-name )
  _Alignof ( type-name )

(6.5.3) unary-operator: one of

\& \* + - !
(6.5.4) cast-expression:
    unary-expression
    ( type-name ) cast-expression

(6.5.5) multiplicative-expression:
    cast-expression
    multiplicative-expression * cast-expression
    multiplicative-expression / cast-expression
    multiplicative-expression % cast-expression

(6.5.6) additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression

(6.5.7) shift-expression:
    additive-expression
    shift-expression ← << additive-expression
    shift-expression → >> additive-expression

(6.5.8) relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression

(6.5.9) equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression

(6.5.10) AND-expression:
    equality-expression
    AND-expression & equality-expression

(6.5.11) exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression

(6.5.12) inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression

(6.5.13) logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression &&& inclusive-OR-expression

(6.5.14) logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression

(6.5.15) conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression

(6.5.16) assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression

(6.5.16) assignment-operator: one of
    = += -= /= %= &= ^= |&|=<<= >>==
(6.5.17) expression:
expression
    assignment-expression
expression , assignment-expression

(6.6) constant-expression:
conditional-expression

A.2.2 Declarations

(6.7) declaration:
declaration-specifiers init-declarator-listopt = ;
    ~~~~~~~~~~~~~~~~~attribute-specifier-sequence declaration-specifiers init-declarator-list ;
    static_assert-declaration
    ~~~~~~~~~~~~~~~~~attribute-declaration

(6.7) declaration-specifiers:
    ~~~~~~~~~~~~~~~~~~~storage-class-specifier declaration-specifiers
    attribute-specifier-sequenceopt
    typedef declaration-specifiers opt declaration-specifier

(6.7) declaration-specifier:
type-qualifier declaration-specifiers opt
    function-specifier declaration-specifiers opt
type-specifier-qualifier
    alignment-specifier declaration-specifiers opt
    function-specifier

(6.7) init-declarator-list:
init-declarator
init-declarator-list , init-declarator

(6.7) init-declarator:
declarator
init-declarator-list , init-declarator

declarator = initializer

(6.7) attribute-declaration:
    ~~~~~~~~~~~~~~~~~attribute-specifier-sequence ;

(6.7.1) storage-class-specifier:
typedef
extern
static
_Thread_local
auto
register

(6.7.2) type-specifier:
void
cchar
short
int
long
float
double
signed
_Bool
_Complex
_Decimal32
_Decimal64
_Decimal128
atomic-type-specifier
struct-or-union-specifier
cenum-specifier
typedef-name
(6.7.2.1) struct-or-union-specifier:
    struct_or_union identifier_opt { struct-declaration-list
  }
(6.7.2.1) struct-or-union_attribute-specifier-sequence_opt identifier_opt { member-declaration-list }

(6.7.2.1) struct-or-union:
    struct
    union

(6.7.2.1) struct-declaration-list member-declaration-list:
    struct-declaration member-declaration
    struct-declaration-list

(6.7.2.1) struct-declaration member-declaration:
    specifier-qualifier-list struct-declarator-list \attribute-specifier-sequence_opt
    static_assert-declaration

(6.7.2.1) specifier-qualifier-list:
    type-specifier specifier-qualifier-list_opt type-specifier

(6.7.2.1) type-specifier-qualifier:
    type-specifier
    \type-qualifier
    alignment-specifier specifier-qualifier-list_opt

(6.7.2.1) struct-declarator-list member-declarator-list:
    struct-declarator member-declarator
    struct-declarator-list

(6.7.2.1) struct-declarator member-declarator:
    declarator_opt : \constant-expression

(6.7.2.2) enum-specifier:
    enum attribute-specifier-sequence_opt identifier_opt { enumerator-list }
    enum attribute-specifier-sequence_opt identifier_opt { enumerator-list , }
    enum identifier

(6.7.2.2) \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\}\textit{\texttt{\textbackslash A.2.2 Language syntax summary}} 403
(6.7.6) **declarator:**

\[ \text{pointer}_{opt} \text{ direct-declarator} \]

(6.7.6) **direct-declarator:**

\[ \text{identifier \_attribute-specifier-sequence}_{opt} \]
\[ \text{( \_declarator \_)} \]
\[ \text{array-declarator \_attribute-specifier-sequence}_{opt} \]
\[ \text{function-declarator \_attribute-specifier-sequence}_{opt} \]

(6.7.6) **array-declarator:**

\[ \text{direct-declarator [ type-qualifier-list}_{opt} \text{ assignment-expression}_{opt} \] \]
\[ \text{direct-declarator [ static type-qualifier-list}_{opt} \text{ assignment-expression } \]
\[ \text{direct-declarator [ type-qualifier-list static assignment-expression ]} \]
\[ \text{direct-declarator [ type-qualifier-list}_{opt} \text{ * } \]
\[ \text{direct-declarator ( \_parameter-type-list \_)} \]

(6.7.6) **function-declarator:**

\[ \text{direct-declarator ( \_identifier-list}_{opt} \text{ parameter-type-list}_{opt} \_ ) \]

(6.7.6) **pointer:**

\[ \text{* \_attribute-specifier-sequence}_{opt} \text{ type-qualifier-list}_{opt} \]
\[ \text{* \_attribute-specifier-sequence}_{opt} \text{ type-qualifier-list}_{opt} \text{ pointer} \]

(6.7.6) **type-qualifier-list:**

\[ \text{type-qualifier} \]
\[ \text{type-qualifier-list type-qualifier} \]

(6.7.6) **parameter-type-list:**

\[ \text{parameter-list} \]
\[ \text{parameter-list , ...} \]

(6.7.6) **parameter-list:**

\[ \text{parameter-declaration} \]
\[ \text{parameter-list , parameter-declaration} \]

(6.7.6) **parameter-declaration:**

\[ \text{attribute-specifier-sequence}_{opt} \text{ declaration-specifiers declarator} \]
\[ \text{attribute-specifier-sequence}_{opt} \text{ declaration-specifiers abstract-declarator}_{opt} \]

(6.7.7) **identifier-list:**

\[ \text{identifier} \]
\[ \text{identifier-list , identifier (6.7.7) type-name:} \]
\[ \text{specifier-qualifier-list} \text{ list-declarator}_{opt} \]

(6.7.7) **abstract-declarator:**

\[ \text{pointer} \]
\[ \text{pointer}_{opt} \text{ direct-abstract-declarator} \]

(6.7.7) **direct-abstract-declarator:**

\[ \text{( \_abstract-declarator \_)} \]
\[ \text{direct-abstract-declarator}_{opt} [\_ \text{type-qualifier-list}_{opt} \_array-abstract-declarator \_attribute-specifier-sequence}_{opt} \]
\[ \text{function-abstract-declarator \_attribute-specifier-sequence}_{opt} \]

(6.7.7) **array-abstract-declarator:**

\[ \text{direct-abstract-declarator}_{opt} [\_ \text{type-qualifier-list}_{opt} \_assignment-expression}_{opt} \]
\[ \text{direct-abstract-declarator}_{opt} [\_ \text{static type-qualifier-list}_{opt} \_assignment-expression ] \]
\[ \text{direct-abstract-declarator}_{opt} \_type-qualifier-list \_static \_assignment-expression ] \]
\[ \text{direct-abstract-declarator}_{opt} [ \_ \text{* } \]

(6.7.7) **function-abstract-declarator:**

\[ \text{direct-abstract-declarator}_{opt} ( \_parameter-type-list}_{opt} \_)

404 Language syntax summary § A.2.2
(6.7.8) typedef-name:
  identifier

(6.7.9) initializer:
  assignment-expression
  \{ initializer-list \}
  \{ initializer-list , \}

(6.7.9) initializer-list:
  designation_opt initializer
  initializer-list , designation_opt initializer

(6.7.9) designation:
  designator-list =

(6.7.9) designator-list:
  designator
  designator-list designator

(6.7.9) designator:
  [ constant-expression ]
  . identifier

(6.7.10) static assert-declaration:
  _Static_assert ( constant-expression , string-literal ) ;
  _Static_assert ( constant-expression ) ;

(6.7.11.1) attribute-specifier-sequence:
  attribute-specifier-sequence_opt attribute-specifier

(6.7.11.1) attribute-specifier:
  \[ attribute-list \] \]

(6.7.11.1) attribute-list:
  attribute_opt
  attribute-list , attribute_opt

(6.7.11.1) attribute:
  attribute-token attribute-argument-clause_opt

(6.7.11.1) attribute-token:
  standard-attribute
  attribute-prefixed-token

(6.7.11.1) standard-attribute:
  identifier

(6.7.11.1) attribute-prefixed-token:
  attribute-prefix :: identifier

(6.7.11.1) attribute-prefix:
  identifier

(6.7.11.1) attribute-argument-clause:
  ( balanced-token-sequence_opt )

(6.7.11.1) balanced-token-sequence:
  balanced-token
  balanced-token-sequence balanced-token

(6.7.11.1) balanced-token:
  ( balanced-token-sequence_opt )
  [ balanced-token-sequence_opt ]
  \{ balanced-token-sequence_opt \}

A.2.3 Statements
(6.8) statement:
  labeled-statement
  compound-statement expression-statement
  expression-statement attribute-specifier-sequence_opt compound-statement
  attribute-specifier-sequence_opt selection-statement
  attribute-specifier-sequence_opt iteration-statement
  attribute-specifier-sequence_opt jump-statement

(6.8.1) labeled-statement:
  attribute-specifier-sequence_opt identifier : statement
  attribute-specifier-sequence_opt case constant-expression : statement
  attribute-specifier-sequence_opt default : statement

(6.8.2) compound-statement:
  { block-item-list_opt }

(6.8.2) block-item-list:
  block-item opt

(6.8.2) block-item:
  block-item-list block-item

(6.8.3) expression-statement:
  expression_opt ;
  attribute-specifier-sequence expression ;

(6.8.4) selection-statement:
  if ( expression ) statement
  if ( expression ) statement else statement
  switch ( expression ) statement

(6.8.5) iteration-statement:
  while ( expression ) statement
  do statement while ( expression ) ;
  for ( expression_opt ; expression_opt ; expression_opt ) statement
  for ( declaration expression_opt ; expression_opt ) statement

(6.8.6) jump-statement:
  goto identifier ;
  continue ;
  break ;
  return expression_opt ;

A.2.4 External definitions
(6.9) translation-unit:
  external-declaration
  translation-unit external-declaration

(6.9) external-declaration:
  function-definition
  declaration

(6.9.1) function-definition:
  declaration-specifiers declarator declaration-list_opt compound-statement
  attribute-specifier-sequence_opt declaration-specifiers declarator function-body
A.3 Preprocessing directives

(6.10) preprocessing-file:
  group_opt
(6.10) group:
  group-part
group group-part
(6.10) group-part:
  if-section
ccontrol-line
text-line
  # # non-directive
(6.10) if-section:
  if-group elif-groups_opt else-group_opt endif-line
(6.10) if-group:
  #if # if constant-expression new-line group_opt
  #ifdef # ifdef identifier new-line group_opt
  #ifndef # ifndef identifier new-line group_opt
(6.10) elif-groups:
  elif-group
elif-groups elif-group
(6.10) elif-group:
  #elif # elif constant-expression new-line group_opt
(6.10) else-group:
  # else # else new-line group_opt
(6.10) endif-line:
  #endif # endif new-line
(6.10) control-line:
  #include # include pp-tokens new-line
  #define # define identifier replacement-list new-line
  #define # define identifier lparen identifier-list_opt )
replacement-list new-line
  #define # define identifier lparen ... )
replacement-list new-line
  #define # define identifier lparen identifier-list .... )
replacement-list new-line
  #undef # undef identifier new-line
  #line # line pp-tokens new-line
  #error # error pp-tokens_opt new-line
  #pragma # pragma pp-tokens_opt new-line
  # # new-line
(6.10) text-line:
  pp-tokens_opt new-line
(6.10) non-directive:
  pp-tokens new-line
(6.10) lparen:
  ( character not immediately preceded by white-space ) character not immediately preceded by white-space
(6.10) replacement-list:
  pp-tokens_opt
(6.10) **pp-tokens:**

`preprocessing-token`

`pp-tokens` `preprocessing-token`

(6.10) **new-line:**

the new-line character

(6.10) **identifier-list:**

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~ identifier
~~~~~~~~~~~~~~~~~~~~~~~~~~~~ identifier-list , identifier
```

(6.10.6) **standard-pragmas:**

```
# pragma STDC FP_CONTRACT on-off-switch
# pragma STDC FENV_ACCESS on-off-switch
# pragma STDC FENV_DEC_ROUND dec-direction
# pragma STDC FENV_ROUND direction
# pragma STDC CX_LIMITED_RANGE on-off-switch
```

(6.10.6) **on-off-switch:** one of

```
ON OFF DEFAULT ON OFF DEFAULT
```

(6.10.6) **direction:** one of

```
FE_DOWNWARD FE_TONEAREST FE_TONEARESTFROMZERO
FE_TOWARDZERO FE_UPWARD FE_DYNAMIC
```

(6.10.6) **dec-direction:** one of

```
FE_DEC_DOWNWARD FE_DEC_TONEAREST FE_DEC_TONEARESTFROMZERO
FE_DEC_TOWARDZERO FE_DEC_UPWARD FE_DEC_DYNAMIC
```

### A.4 Floating-point subject sequence

#### A.4.1 NaN char sequence

(7.22.1.5) **n-char-sequence:**

```
digit
nondigit
```

n-char-sequence `digit`

n-char-sequence `nondigit`

#### A.4.2 NaN wchar_t sequence

(7.29.4.1.1) **n-wchar-sequence:**

```
digit
nondigit
```

n-wchar-sequence `digit`

n-wchar-sequence `nondigit`

### A.5 Decimal floating-point subject sequence

#### A.5.1 NaN decimal char sequence

(7.22.1.6) **d-char-sequence:**

```
digit
nondigit
```

d-char-sequence `digit`

d-char-sequence `nondigit`

#### A.5.2 NaN decimal wchar_t sequence
\[ (7.29.4.1.2) \]
\[
\begin{align*}
\text{d-wchar-sequence:} & \\
\text{digit} & \\
\text{nondigit} & \\
\text{d-wchar-sequence digit} & \\
\text{d-wchar-sequence nondigit} & 
\end{align*}
\]
Annex B
(informative)
Library summary

B.1 Diagnostics <assert.h>

```c
#include <assert.h>

NDEBUG static_assert;

void assert(scalar expression);
```

B.2 Complex <complex.h>

```c
#pragma STDC CX_LIMITED_RANGE on-off-switch

complex __STDC_NO_COMPLEX__ imaginary

_complex_I Imaginary_I

I

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);

cos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

cosh(double complex z);
float complex ccosh(float complex z);
long double complex ccoshl(long double complex z);

coshl(long double complex z);
float complex csinh(float complex z);
long double complex csinh(float complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```
long double complex clogl(long double complex z);
double cabs(double complex z);
float cabsf(float complex z);
long double complex clog(long double complex z);
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
double carg(double complex z);
float cargf(float complex z);
long double complex cargl(long double complex z);
double cimag(double complex z);
float cimagf(float complex z);
long double complex cimagl(long double complex z);
double complex CMPLX(double x, double y);
float complex CMPLXF(float x, float y);
long double complex CMPLXL(long double x, long double y);
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
Only if the implementation defines `__STDC_IEC_60559_DFP__`:

```
FE_DEC_DOWNWARD   FE_DEC_TONEARESTFROMZERO   FE_DEC_UPWARD
FE_DEC_TONEAREST  FE_DEC_TOWARDZERO
```

```
#pragma STDC FENV_DEC_ROUND dec-direction
int fe_dec_getround(void);
int fe_dec_setround(int round);
```

### B.6 Characteristics of floating types `<float.h>`

- `FLT_ROUNDS`
- `FLT_EVAL_METHOD`
- `FLT_HAS_SUBNORM`
- `DBL_HAS_SUBNORM`
- `LDBL_HAS_SUBNORM`
- `FLT_RADIX`
- `FLT_MANT_DIG`
- `DBL_MANT_DIG`
- `LDBL_MANT_DIG`
- `DECIMAL_DIG`
- `FLT_DECIMAL_DIG`
- `DBL_DECIMAL_DIG`
- `LDBL_DECIMAL_DIG`
- `FLT_MAXEXP`
- `DBL_MAXEXP`
- `LDBL_MAXEXP`
- `FLT_MINEXP`
- `DBL_MINEXP`
- `LDBL_MINEXP`
- `FLT_MAX_10_EXP`
- `DBL_MAX_10_EXP`
- `LDBL_MAX_10_EXP`
- `FLT_MAX`
- `DBL_MAX`
- `LDBL_MAX`
- `FLT_EPSILON`
- `DBL_EPSILON`
- `LDBL_EPSILON`
- `FLT_TRUE_MIN`
- `DBL_TRUE_MIN`
- `LDBL_TRUE_MIN`

### B.6.1 Characteristics of decimal floating types

The following macros are provided only if the implementation defines `__STDC_IEC_60559_DFP__`. `N` is 32, 64 and 128.
B.7 Format conversion of integer types <inttypes.h>

\begin{verbatim}
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
\end{verbatim}

B.8 Alternative spellings <iso646.h>

\begin{verbatim}
and and_eq bitand compl not not_eq xor xor_eq
bitand not or or_eq
\end{verbatim}

B.9 Sizes of integer types <limits.h>

\begin{verbatim}
CHAR_BIT CHAR_MAX INT_MIN ULONG_MAX
SCHAR_MIN MB_LEN_MAX INT_MAX LONGLONG_MIN
SCHAR_MAX SHRT_MIN UINT_MAX LONGLONG_MAX
UCHAR_MAX SHRT_MAX LONG_MIN ULONGLONG_MAX
CHAR_MIN USHRT_MAX LONG_MAX
\end{verbatim}

B.10 Localization <locale.h>

\begin{verbatim}
struct lconv LC_ALL LC_COLLATE LC_CTYPE LC_NUMERIC LC_TIME
NULL LC_MONETARY LC_MONETARY LC_TIME
\end{verbatim}

\begin{verbatim}
char * setlocale(int category, const char * locale);
struct lconv *localeconv(void);
\end{verbatim}

B.11 Mathematics <math.h>

\begin{verbatim}
float_t HUGE_VAL float_t FP_NAN
double_t INFINITY double_t FP_NORMAL
HUGE_VAL NAN double_t FP_SUBNORMAL
HUGE_VALF FP_INFINITE double_t FP_ZERO
\end{verbatim}
#pragma STDC FP_CONTRACT on-off-switch

int fpclassify(real-floating x);
int iscanonical(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);
int signbit(real-floating x);
int issignaling(real-floating x);
int isnormal(real-floating x);

double acos(double x);
float acosf(float x);
long double acosl(long double x);

double acosl(real-floating x);
float acosl(real-floating x);
long double acosl(real-floating x);

double atan(double x);
float atanf(float x);
long double atanl(long double x);

double atan(real-floating x);
float atanf(real-floating x);
long double atanl(real-floating x);

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

double atan2(real-floating x);
float atan2f(real-floating x);
long double atan2l(real-floating x);

double cos(double x);
float cosf(float x);
long double cosl(long double x);

double cos(real-floating x);
float cosf(real-floating x);
long double cosl(real-floating x);

double sin(double x);
float sinf(float x);
long double sinl(long double x);

double sin(real-floating x);
float sinf(real-floating x);
long double sinl(real-floating x);

double tan(double x);
float tanf(float x);
long double tanl(long double x);

double tan(real-floating x);
float tanf(real-floating x);
long double tanl(real-floating x);

double asin(double x);
float asinf(float x);
long double asinl(long double x);

double asin(real-floating x);
float asinf(real-floating x);
long double asinl(real-floating x);

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);

double atan2(real-floating x);
float atan2f(real-floating x);
long double atan2l(real-floating x);
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
double exp(double x);
float expf(float x);
long double expl(long double x);
double exp10(double x);
float exp10f(float x);
long double exp10l(long double x);
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
double exp10m1(double x);
float exp10m1f(float x);
long double exp10m1l(long double x);
float frexp(float value, int *p);
long double frexpl(long double value, int *p);
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
double ldexp(double x, int p);
float ldexpf(float x, int p);
long double ldexpl(long double x, int p);
long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
double log(double x);
float logf(float x);
long double logl(long double x);
double log10(double x);
float log10f(float x);
long double log10l(long double x);
double log10p1(double x);
float log10p1f(float x);
long double log10p1l(long double x);
float log1p(float x);
long double log1pl(long double x);
double log2(double x);
float log2f(float x);
long double log2l(long double x);
double log2p1(double x);
float log2p1f(float x);
long double log2p1l(long double x);
double logb(double x);
float logbf(float x);
long double logbl(long double x);
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, long int n);
double cbrt(double x);
float cbrtf(float x);
long double cbartl(long double x);
double compoundn(double x, intmax_t n);
float compoundnf(float x, intmax_t n);
long double compoundnl(long double x, intmax_t n);
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
double pown(double x, intmax_t n);
float pownf(float x, intmax_t n);
long double pownl(long double x, intmax_t n);
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
double erf(double x);
float erff(float x);
long double erfll(long double x);
double erfc(double x);
float erfcf(float x);
long double erfcfl(long double x);
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
double floor(double x);
float floorf(float x);
long double floorl(long double x);
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
double rint(double x);
float rintf(float x);
long double rintl(long double x);
float lrintf(float x);
long int lrint(long double x);
long int lrintf(float x);
long long int llrint(long double x);
long long int llrintf(float x);
long double round(double x);
float roundf(float x);
long double roundl(long double x);
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(long double x);
long long int llroundf(float x);
long long int llroundl(long double x);
float roundeven(float x);
long double roundevenl(long double x);
long int lroundeven(double x);
float truncf(float x);
long double truncl(long double x);
float truncf(float x);
long double trunc(long double x);
intmax_t fromfpf(double x, int round, unsigned int width);
uintmax_t ufromfpf(double x, int round, unsigned int width);
uintmax_t ufromfpf(float x, int round, unsigned int width);
uintmax_t ufromfpf(long double x, int round, unsigned int width);
uintmax_t ufromfpf(float x, int round, unsigned int width);
uintmax_t ufromfpf(long double x, int round, unsigned int width);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
double remquof(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
double nextafter(double x, double y);
float nextafterf(float x, float y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);
double nextdown(double x);
float nextdownf(float x);
long double nextdownl(long double x);
int canonicalize(double * cx, const double * x);
int canonicalizef(float * cx, const float * x);
int canonicalizel(long double * cx, const long double * x);
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
double fmaxmag(double x, double y);
float fmaxmagf(float x, float y);
long double fmaxmagl(long double x, long double y);
double fminmag(double x, double y);
float fminmagf(float x, float y);
long double fminmagl(long double x, long double y);
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
float fadd(double x, double y);
float faddl(long double x, long double y);
double faddl(long double x, long double y);
float fsubl(double x, double y);
float fmul(double x, long double y);
long double fmull(long double x, long double y);
float fdiv(double x, double y);
float fdivl(long double x, long double y);
double ddivl(long double x, long double y);
float fmal(long double x, double y, double z);
float fmal(long double x, long double y, long double z);
double dfmal(long double x, long double y, long double z);
float fsqrtl(long double x);
float dsqrtl(long double x);
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islesssgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
int iseqsig(real-floating x, real-floating y);

_only if the implementation defines __STDC_IEC_60559_DFP__: _Decimal32 acosd32(_Decimal32 x);
Decimal64 acosd64(_Decimal64 x);
Decimal128 acosd128(_Decimal128 x);
- _Decimal32 asind32(_Decimal32 x);
- _Decimal64 asind64(_Decimal64 x);
- _Decimal128 asind128(_Decimal128 x);
- _Decimal32 atand32(_Decimal32 x);
- _Decimal64 atand64(_Decimal64 x);
- _Decimal128 atand128(_Decimal128 x);
- _Decimal32 atan2d32(_Decimal32 x, _Decimal32 y);
- _Decimal64 atan2d64(_Decimal64 x, _Decimal64 y);
- _Decimal128 atan2d128(_Decimal128 x, _Decimal128 y);
- _Decimal32 cosd32(_Decimal32 x);
- _Decimal64 cosd64(_Decimal64 x);
- _Decimal128 cosd128(_Decimal128 x);
- _Decimal32 sind32(_Decimal32 x);
- _Decimal64 sind64(_Decimal64 x);
- _Decimal128 sind128(_Decimal128 x);
- _Decimal32 tand32(_Decimal32 x);
- _Decimal64 tand64(_Decimal64 x);
- _Decimal128 tand128(_Decimal128 x);
- _Decimal32 acospid32(_Decimal32 x);
- _Decimal64 acospid64(_Decimal64 x);
- _Decimal128 acospid128(_Decimal128 x);
- _Decimal32 asinpid32(_Decimal32 x);
- _Decimal64 asinpid64(_Decimal64 x);
- _Decimal128 asinpid128(_Decimal128 x);
- _Decimal32 atanpid32(_Decimal32 x);
- _Decimal64 atanpid64(_Decimal64 x);
- _Decimal128 atanpid128(_Decimal128 x);
- _Decimal32 atan2pid32(_Decimal32 y, _Decimal32 x);
- _Decimal64 atan2pid64(_Decimal64 y, _Decimal64 x);
- _Decimal128 atan2pid128(_Decimal128 y, _Decimal128 x);
- _Decimal32 cospid32(_Decimal32 x);
- _Decimal64 cospid64(_Decimal64 x);
- _Decimal128 cospid128(_Decimal128 x);
- _Decimal32 sinpid32(_Decimal32 x);
- _Decimal64 sinpid64(_Decimal64 x);
- _Decimal128 sinpid128(_Decimal128 x);
- _Decimal32 tanpid32(_Decimal32 x);
- _Decimal64 tanpid64(_Decimal64 x);
- _Decimal128 tanpid128(_Decimal128 x);
- _Decimal32 acoshd32(_Decimal32 x);
- _Decimal64 acoshd64(_Decimal64 x);
- _Decimal128 acoshd128(_Decimal128 x);
- _Decimal32 asinhd32(_Decimal32 x);
- _Decimal64 asinhd64(_Decimal64 x);
- _Decimal128 asinhd128(_Decimal128 x);
- _Decimal32 tanhd32(_Decimal32 x);
- _Decimal64 tanhd64(_Decimal64 x);
- _Decimal128 tanhd128(_Decimal128 x);
- _Decimal32 expd32(_Decimal32 x);
- _Decimal64 expd64(_Decimal64 x);
- _Decimal128 expd128(_Decimal128 x);
- _Decimal32 exp10d32(_Decimal32 x);
\begin{itemize}
\item \texttt{Decimal64 exp10d64(Decimal64 x)};
\item \texttt{Decimal128 exp10d128(Decimal128 x)};
\item \texttt{Decimal32 exp10m32(Decimal32 x)};
\item \texttt{Decimal64 exp10m64(Decimal64 x)};
\item \texttt{Decimal128 exp10m128(Decimal128 x)};
\item \texttt{Decimal32 exp2d32(Decimal32 x)};
\item \texttt{Decimal64 exp2d64(Decimal64 x)};
\item \texttt{Decimal128 exp2d128(Decimal128 x)};
\item \texttt{Decimal32 expm1d32(Decimal32 x)};
\item \texttt{Decimal64 expm1d64(Decimal64 x)};
\item \texttt{Decimal128 expm1d128(Decimal128 x)};
\item \texttt{Decimal32 frexpd32(Decimal32 value, int* p)};
\item \texttt{Decimal64 frexpd64(Decimal64 value, int* p)};
\item \texttt{Decimal128 frexpd128(Decimal128 value, int* p)};
\item \texttt{int ilogbd32(Decimal32 x)};
\item \texttt{int ilogbd64(Decimal64 x)};
\item \texttt{int ilogbd128(Decimal128 x)};
\item \texttt{Decimal32 logd32(Decimal32 x)};
\item \texttt{Decimal64 logd64(Decimal64 x)};
\item \texttt{Decimal128 logd128(Decimal128 x)};
\item \texttt{Decimal32 log10d32(Decimal32 x)};
\item \texttt{Decimal64 log10d64(Decimal64 x)};
\item \texttt{Decimal128 log10d128(Decimal128 x)};
\item \texttt{Decimal32 log2d32(Decimal32 x)};
\item \texttt{Decimal64 log2d64(Decimal64 x)};
\item \texttt{Decimal128 log2d128(Decimal128 x)};
\item \texttt{Decimal32 modfd32(Decimal32 x, Decimal32* iptr)};
\item \texttt{Decimal64 modfd64(Decimal64 x, Decimal64* iptr)};
\item \texttt{Decimal128 modfd128(Decimal128 x, Decimal128* iptr)};
\item \texttt{Decimal32 scalbnd32(Decimal32 x, int n)};
\item \texttt{Decimal64 scalbnd64(Decimal64 x, int n)};
\item \texttt{Decimal128 scalbnd128(Decimal128 x, int n)};
\item \texttt{Decimal32 scalblnd32(Decimal32 x, long int n)};
\item \texttt{Decimal64 scalblnd64(Decimal64 x, long int n)};
\item \texttt{Decimal128 scalblnd128(Decimal128 x, long int n)};
\item \texttt{Decimal32 cbrtd32(Decimal32 x)};
\item \texttt{Decimal64 cbrtd64(Decimal64 x)};
\end{itemize}
_Decimal128 cbrtd128(_Decimal128 x);

_Decimal32 compoundnd32(_Decimal32 x, intmax_t n);
_Decimal64 compoundnd64(_Decimal64 x, intmax_t n);
_DecIMAL128 compoundnd128(_Decimal128 x, intmax_t n);

_DecIMAL32 fabsd32(_Decimal32 x);

_DecIMAL64 fabsd64(_Decimal64 x);

_DecIMAL128 fabsd128(_Decimal128 x);

_DecIMAL32 hypotd32(_Decimal32 x, _Decimal32 y);

_DecIMAL64 hypotd64(_Decimal64 x, _Decimal64 y);

_DecIMAL128 hypotd128(_Decimal128 x, _Decimal128 y);

_DecIMAL32 powd32(_Decimal32 x, _Decimal32 y);

_DecIMAL64 powd64(_Decimal64 x, _Decimal64 y);

_DecIMAL128 powd128(_Decimal128 x, _Decimal128 y);

_DecIMAL32 pownd32(_Decimal32 x, intmax_t n);

_DecIMAL64 pownd64(_Decimal64 x, intmax_t n);

_DecIMAL128 pownd128(_Decimal128 x, intmax_t n);

_DecIMAL32 powrd32(_Decimal32 y, _Decimal32 x);

_DecIMAL64 powrd64(_Decimal64 y, _Decimal64 x);

_DecIMAL128 powrd128(_Decimal128 y, _Decimal128 x);

_DecIMAL32 rootnd32(_Decimal32 x, intmax_t n);

_DecIMAL64 rootnd64(_Decimal64 x, intmax_t n);

_DecIMAL128 rootnd128(_Decimal128 x, intmax_t n);

_DecIMAL32 rsqrtd32(_Decimal32 x);

_DecIMAL64 rsqrtd64(_Decimal64 x);

_DecIMAL128 rsqrtd128(_Decimal128 x);

_DecIMAL32 sqrt32(_Decimal32 x);

_DecIMAL64 sqrt64(_Decimal64 x);

_DecIMAL128 sqrt128(_Decimal128 x);

_DecIMAL32 erf32(_Decimal32 x);

_DecIMAL64 erf64(_Decimal64 x);

_DecIMAL128 erf128(_Decimal128 x);

_DecIMAL32 erfc32(_Decimal32 x);

_DecIMAL64 erfc64(_Decimal64 x);

_DecIMAL128 erfc128(_Decimal128 x);

_DecIMAL32 lgammad32(_Decimal32 x);

_DecIMAL64 lgammad64(_Decimal64 x);

_DecIMAL128 lgammad128(_Decimal128 x);

_DecIMAL32 tgammad32(_Decimal32 x);

_DecIMAL64 tgammad64(_Decimal64 x);

_DecIMAL128 tgammad128(_Decimal128 x);

_DecIMAL32 ceil32(_Decimal32 x);

_DecIMAL64 ceil64(_Decimal64 x);

_DecIMAL128 ceil128(_Decimal128 x);

_DecIMAL32 floor32(_Decimal32 x);

_DecIMAL64 floor64(_Decimal64 x);

_DecIMAL128 floor128(_Decimal128 x);

_DecIMAL32 nearbyint32(_Decimal32 x);

_DecIMAL64 nearbyint64(_Decimal64 x);

_DecIMAL128 nearbyint128(_Decimal128 x);

_DecIMAL32 rint32(_Decimal32 x);

_DecIMAL64 rint64(_Decimal64 x);

_DecIMAL128 rint128(_Decimal128 x);

long int lrint32(_Decimal32 x);

long int lrint64(_Decimal64 x);

long int lrint128(_Decimal128 x);

long long int llrint32(_Decimal32 x);

long long int llrint64(_Decimal64 x);

long long int llrint128(_Decimal128 x);

_DecIMAL32 round32(_Decimal32 x);

_DecIMAL64 round64(_Decimal64 x);

_DecIMAL128 round128(_Decimal128 x);
long int lroundd32(_Decimal32 x);
long int lroundd64(_Decimal64 x);
long int lroundd128(_Decimal128 x);
long long int llroundd32(_Decimal32 x);
long long int llroundd64(_Decimal64 x);
long long int llroundd128(_Decimal128 x);
 Decimal128 roundevend32(_Decimal128 x);
 Decimal64 roundevend64(_Decimal64 x);
 Decimal128 roundevend128(_Decimal128 x);
 Decimal32 roundevend32(_Decimal32 x);
 Decimal64 roundevend64(_Decimal64 x);
 Decimal128 roundevend128(_Decimal128 x);
 Decimal32 truncd32(_Decimal32 x);
 Decimal64 truncd64(_Decimal64 x);
 Decimal128 truncd128(_Decimal128 x);
 Decimal32 roundeven_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 roundeven_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 roundeven_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 remainderd32(_Decimal32 x, _Decimal32 y);
 Decimal64 remainderd64(_Decimal64 x, _Decimal64 y);
 Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);
 Decimal32 copysign_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 copysign_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 copysign_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 nand32(const char *tagp);
 Decimal64 nand64(const char *tagp);
 Decimal128 nand128(const char *tagp);
 Decimal32 nextafter_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 nextafter_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 nextafter_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 nexttoward_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 nexttoward_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 nexttoward_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 nextup_d32(_Decimal32 x);
 Decimal64 nextup_d64(_Decimal64 x);
 Decimal128 nextup_d128(_Decimal128 x);
 Decimal32 nextdown_d32(_Decimal32 x);
 Decimal64 nextdown_d64(_Decimal64 x);
 Decimal128 nextdown_d128(_Decimal128 x);
 int canonicalized_d32(_Decimal32 cx, const _Decimal32 *x);
 int canonicalized_d64(_Decimal64 cx, const _Decimal64 *x);
 int canonicalized_d128(_Decimal128 cx, const _Decimal128 *x);
 Decimal32 fdim_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 fdim_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 fdim_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 fmax_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 fmax_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 fmax_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 fmin_d32(_Decimal32 x, _Decimal32 y);
 Decimal64 fmin_d64(_Decimal64 x, _Decimal64 y);
 Decimal128 fmin_d128(_Decimal128 x, _Decimal128 y);
 Decimal32 fmaxmag_d32(_Decimal32 x, _Decimal32 y);
### B.11 Library summary

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Decimal64 fmaxmagd64(_Decimal64 x, _Decimal64 y)</code></td>
<td>Maximum magnitude of two <code>Decimal64</code> values.</td>
</tr>
<tr>
<td><code>Decimal32 fminmagd32(_Decimal32 x, _Decimal32 y)</code></td>
<td>Minimum magnitude of two <code>Decimal32</code> values.</td>
</tr>
<tr>
<td><code>Decimal64 fminmagd64(_Decimal64 x, _Decimal64 y)</code></td>
<td>Minimum magnitude of two <code>Decimal64</code> values.</td>
</tr>
<tr>
<td><code>Decimal64 fmad64(_Decimal64 x, _Decimal64 y)</code></td>
<td>Multiplication and addition of two <code>Decimal64</code> values.</td>
</tr>
<tr>
<td><code>Decimal32 fmad32(_Decimal32 x, _Decimal32 y)</code></td>
<td>Multiplication and addition of two <code>Decimal32</code> values.</td>
</tr>
<tr>
<td><code>Bool samequantumd128(_Decimal128 x, _Decimal128 y)</code></td>
<td>Checks if two <code>Decimal128</code> values are quantized.</td>
</tr>
<tr>
<td><code>Bool samequantumd64(_Decimal64 x, _Decimal64 y)</code></td>
<td>Checks if two <code>Decimal64</code> values are quantized.</td>
</tr>
<tr>
<td><code>Bool samequantumd32(_Decimal32 x, _Decimal32 y)</code></td>
<td>Checks if two <code>Decimal32</code> values are quantized.</td>
</tr>
<tr>
<td><code>void decodebind128(unsigned char *xptr, restrict static 4)</code></td>
<td>Decodes a <code>Decimal128</code> value.</td>
</tr>
<tr>
<td><code>void decodebind64(unsigned char *xptr, restrict static 8)</code></td>
<td>Decodes a <code>Decimal64</code> value.</td>
</tr>
<tr>
<td><code>void decodebind32(unsigned char *xptr, restrict static 4)</code></td>
<td>Decodes a <code>Decimal32</code> value.</td>
</tr>
<tr>
<td><code>void encodebind128(unsigned char *xptr, restrict static 4)</code></td>
<td>Encodes a <code>Decimal128</code> value.</td>
</tr>
<tr>
<td><code>void encodebind64(unsigned char *xptr, restrict static 8)</code></td>
<td>Encodes a <code>Decimal64</code> value.</td>
</tr>
<tr>
<td><code>void encodebind32(unsigned char *xptr, restrict static 4)</code></td>
<td>Encodes a <code>Decimal32</code> value.</td>
</tr>
<tr>
<td><code>void decodedecd128(unsigned char *xptr, restrict static 4)</code></td>
<td>Decodes a <code>Decimal128</code> value.</td>
</tr>
<tr>
<td><code>void decodedecd64(unsigned char *xptr, restrict static 8)</code></td>
<td>Decodes a <code>Decimal64</code> value.</td>
</tr>
<tr>
<td><code>void decodedecd32(unsigned char *xptr, restrict static 4)</code></td>
<td>Decodes a <code>Decimal32</code> value.</td>
</tr>
<tr>
<td><code>void encodedecd128(unsigned char *xptr, restrict static 4)</code></td>
<td>Encodes a <code>Decimal128</code> value.</td>
</tr>
<tr>
<td><code>void encodedecd64(unsigned char *xptr, restrict static 8)</code></td>
<td>Encodes a <code>Decimal64</code> value.</td>
</tr>
<tr>
<td><code>void encodedecd32(unsigned char *xptr, restrict static 4)</code></td>
<td>Encodes a <code>Decimal32</code> value.</td>
</tr>
<tr>
<td><code>long long int llquantexpd32(_Decimal32 x)</code></td>
<td>Quantizes a <code>Decimal32</code> value.</td>
</tr>
<tr>
<td><code>long long int llquantexpd64(_Decimal64 x)</code></td>
<td>Quantizes a <code>Decimal64</code> value.</td>
</tr>
<tr>
<td><code>long long int llquantexpd128(_Decimal128 x)</code></td>
<td>Quantizes a <code>Decimal128</code> value.</td>
</tr>
</tbody>
</table>
const unsigned char encptr[restrict static 16]);

Only if the implementation defines __STDC_IEC_60559_BFP__ or __STDC_IEC_559__ and additionally the user code defines __STDC_WANT_IEC_60559_BFP_EXT__ before any inclusion of <math.h>:

```c
int totalorder(const double *x, const double *y);
int totalorderf(const float *x, const float *y);
int totalorderl(const long double *x, const long double *y);
int totalordermag(const double *x, const double *y);
int totalordermagf(const float *x, const float *y);
int totalordermagl(const long double *x, const long double *y);
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
```

Only if the implementation defines __STDC_IEC_60559_DFP__ and additionally the user code defines __STDC_WANT_IEC_60559_DFP_EXT__ before any inclusion of <math.h>:

```c
int totalorderd32(const _Decimal32 *x, const _Decimal32 *y);
int totalorderd64(const _Decimal64 *x, const _Decimal64 *y);
int totalorderd128(const _Decimal128 *x, const _Decimal128 *y);
int totalordermagd32(const _Decimal32 *x, const _Decimal32 *y);
int totalordermagd64(const _Decimal64 *x, const _Decimal64 *y);
int totalordermagd128(const _Decimal128 *x, const _Decimal128 *y);
_Decimal32 getpayloadd32(_Decimal32 *x);
_Decimal64 getpayloadd64(_Decimal64 *x);
_Decimal128 getpayloadd128(_Decimal128 *x);
int setpayloadd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadd128(_Decimal128 *res, _Decimal128 pl);
int setpayloadsigd32(_Decimal32 *res, _Decimal32 pl);
int setpayloadsigd64(_Decimal64 *res, _Decimal64 pl);
int setpayloadsigd128(_Decimal128 *res, _Decimal128 pl);
```

B.12 Nonlocal jumps <setjmp.h>

```c
jmp_buf

int setjmp(jmp_buf env);
_Noreturn void longjmp(jmp_buf env, int val);
```

B.13 Signal handling <signal.h>

```c
sig_atomic_t SIG_DFL SIG_ERR SIG_IGN SIGINT SIGTERM
SIG_FPE SIG_TERM SIG_ABRT SIG_USR

void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```
B.14 Alignment <stdalign.h>
alignas alignof __alignas_is_defined __alignof_is_defined

B.15 Variable arguments <stdarg.h>

```c
va_list

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, parmN);
```

B.16 Atomics <stdatomic.h>

```c
#define ATOMIC_VAR_INIT(C value)
void atomic_init(volatile A *obj, C value);
type kill_dependency(type y);
void atomic_thread_fence(memory_order order);
void atomic_signal_fence(memory_order order);
_Bool atomic_is_lock_free(const volatile A *obj);
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
C atomic_load(const volatile A *object);
C atomic_load_explicit(const volatile A *object, memory_order order);
C atomic_exchange(volatile A *object, C desired);
C atomic_exchange_explicit(volatile A *object, C desired, memory_order order);
_Bool atomic_compare_exchange_strong(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(volatile A *object, C *expected, C desired, memory_order success, memory_order failure);
_Bool atomic_compare_exchange_weak(volatile A *object, C *expected, C desired);
_Bool atomic_compare_exchange_weak_explicit(volatile A *object, C *expected, C desired, memory_order order failure);
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
_Bool atomic_flag_test_and_set(volatile atomic_flag *object);
_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object, memory_order order);
void atomic_flag_clear(volatile atomic_flag *object);
```
void atomic_flag_clear_explicit(volatile atomic_flag *object, 
    memory_order order);

B.17 Boolean type and values <stdbool.h>
bool true false __bool_true_false_are_defined

B.18 Common definitions <stddef.h>
ptrdiff_t size_t max_align_t wchar_t NULL

offsetof(type, member-designator)

Only if the implementation defines __STDC_LIB_EXT1__ and additionally the user code defines 
__STDC_WANT_LIB_EXT1__ before any inclusion of <stddef.h>:

rsize_t

B.19 Integer types <stdint.h>

intN_t UINT_LEASTN_MAX PTRDIFF_MAX
uintN_t UINT_LEASTN_WIDTH SIG_ATOMIC_MIN
int_leastN_t INT_FASTN_MIN SIG_ATOMIC_MAX
uint_leastN_t INT_FASTN_MAX SIZE_MAX
int_fastN_t INT_FASTN_WIDTH SIZE_WIDTH
uint_fastN_t INT_FASTN_MAX SIZE_WIDTH
intptr_t UINT_FASTN_WIDTH WCHAR_MIN
uintptr_t INTPTR_MIN WCHAR_MAX
intmax_t INTPTR_MAX WCHAR_WIDTH
uintmax_t INTPTR_WIDTH WINT_MIN
INTN_MIN UINTPTR_MAX WINT_MAX
INTN_MAX UINTPTR_WIDTH WINT_WIDTH
INTN_WIDTH INTMAX_MIN INTN_C(value)
UINTN_MAX INTMAX_MAX UINTN_C(value)
UINTN_WIDTH INTMAX_WIDTH INTMAX_C(value)
INT_LEASTN_MIN UINTMAX_MAX
INT_LEASTN_MAX UINTMAX_WIDTH
INT_LEASTN_WIDTH PTRDIFF_MIN

Only if the implementation defines __STDC_LIB_EXT1__ and additionally the user code defines 
__STDC_WANT_LIB_EXT1__ before any inclusion of <stdint.h>:

RSIZE_MAX

B.20 Input/output <stdio.h>

size_t FILE_NULL _IOFBF _IOLBF FILENAME_MAX TMP_MAX
fpos_t FILE_NULL _IONBF _IOLBF _tmpnam l_tmpnam stderr
FILE_NULL BUFSIZ SEEK_CUR stdin
NULL FILE_NULL SEEK_END stdout
_IOFBF FILE_NULL SEEK_SET

int remove(const char *filename);
int rename(const char *old, const char *new);

426 Library summary § B.20
Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<stdio.h>`:

```
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char * restrict filename, const char * restrict mode);
FILE *freopen(const char * restrict filename, const char * restrict mode,
              FILE * restrict stream);
void setbuf(FILE * restrict stream, char * restrict buf);
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);
int printf(const char * restrict format, ...);
int scanf(const char * restrict format, ...);
int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
int sscn(const char *, restrict s, const char *, restrict format, ...);
int vprintf(FILE * restrict stream, const char * restrict format, va_list arg);
int vsprintf(FILE * restrict stream, const char * restrict format, va_list arg);
int vsprintf(const char * restrict format, va_list arg);
int vsnfprintf(const char *, size_t n, const char *, restrict format, va_list arg);
int vsprintf(const char *, restrict format, va_list arg);
int ssprintf(const char *, size_t n, const char *, restrict format, va_list arg);
int vsprintf(const char *, restrict format, va_list arg);
int fgetpos(FILE *stream);
char *fgets(char *, int n, FILE * restrict stream);
int fputc(int c, FILE *stream);
int getc(FILE *stream);
int getchar(void);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void * restrict ptr, size_t size, size_t nmemb,
             FILE * restrict stream);
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,
              FILE * restrict stream);
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
int fprintf(FILE * restrict stream, const char * restrict format, ...);
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

<table>
<thead>
<tr>
<th>L_tmpnam_s</th>
<th>TMP_MAX_S</th>
<th>errno_t</th>
<th>rsize_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>errno_t tmpfile_s(FILE * restrict * restrict streamptr);</td>
<td>errno_t tmpnam_s(char *, rsize_t maxsize);</td>
<td>errno_t fopen_s(FILE * restrict * restrict streamptr, const char * restrict filename, const char * restrict mode);</td>
<td>errno_t freopen_s(FILE * restrict * restrict newstreamptr, const char * restrict filename, const char * restrict mode, FILE * restrict stream);</td>
</tr>
</tbody>
</table>
B.21 General utilities <stdlib.h>

<table>
<thead>
<tr>
<th>size_t</th>
<th>div_t</th>
<th>lldiv_t</th>
<th>EXIT_FAILURE</th>
<th>RAND_MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>wchar_t</td>
<td>div_t</td>
<td>NULL</td>
<td>EXIT_SUCCESS</td>
<td>MB_CUR_MAX</td>
</tr>
</tbody>
</table>

double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
long long int atoll(const char *nptr);
int strftime(char *restrict s, size_t n, const char *restrict format, double *fp);
int strftime(char *restrict s, size_t n, const char *restrict format, float *fp);
int strftime(char *restrict s, size_t n, const char *restrict format, long double *fp);

double strtod(const char *restrict nptr, char **restrict endptr);
float strftime(const char *restrict nptr, char **restrict endptr);
long double strtooled(const char *restrict nptr, char **restrict endptr);
long int strtol(const char *restrict nptr, char **restrict endptr, int base);
unsigned long int strtoul(const char *restrict nptr, char **restrict endptr, int base);
unsigned long int strtoull(const char *restrict nptr, char **restrict endptr, int base);
int rand(void);
void srand(unsigned int seed);
void *aligned_alloc(size_t alignment, size_t size);
void *calloc(size_t nmemb, size_t size);
void *free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
_return void abort(void);
int atexit(void (*)(void))(void);
int at_quick_exit(void (*)(void))(void);
_return void exit(int status);
_return void _Exit(int status);
char *getenv(const char *name);
_Return void quick_exit(int status);
int system(const char *string);
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(const void *, const void *));
int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
Only if the implementation defines `__STDC_IEC_60559_DFP__`:

```
int strfromd32(char *restrict s, size_t n, const char *restrict format, _Decimal32 fp);
int strfromd64(char *restrict s, size_t n, const char *restrict format, _Decimal64 fp);
int strfromd128(char *restrict s, size_t n, const char *restrict format, _Decimal128 fp);
```

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<stdlib.h>`:

```
errno_t rsize_t constraint_handler_t
```

```c
constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
void abort_handler_s(const char * restrict msg, void * restrict ptr,
    errno_t error);
void ignore_handler_s(const char * restrict msg, void * restrict ptr,
    errno_t error);
errno_t getenv_s(size_t * restrict len, char * restrict value, rsize_t maxsize,
    const char * restrict name);
void *bsearch_s(const void *key, const void *base, rsize_t nmemb, rsize_t size,
    int (*compar)(const void *k, const void *y, void *context),
    void *context);
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
    int (*compar)(const void *x, const void *y, void *context),
    void *context);
errno_t wctomb_s(int *restrict status, char * restrict s, rsize_t smax,
    wchar_t *wc);
errno_t mbstowcs_s(size_t * restrict retval, wchar_t * restrict dst,
    const char * restrict src, size_t len);
errno_t wcstombs_s(size_t * restrict retval, char * restrict dst, size_t dstmax,
    const wchar_t * restrict src, size_t len);
```

### B.22 Noreturn `<stdnoreturn.h>`

```c
noreturn
```

### B.23 String handling `<string.h>`

```
size_t NULL
```

```c
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memccpy(void * restrict s1, const void * restrict s2, int c, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strncpy(char * restrict s1, const char * restrict s2);
char *strncat(char * restrict s1, const char * restrict s2);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
```
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
void * memchr(const void *s, int c, size_t n);
char * strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char * strpbrk(const char *s1, const char *s2);
char * strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char * strstr(const char *s1, const char *s2);
char * strtok(char * restrict s1, const char * restrict s2);
void * memset(void *s, int c, size_t n);
char * strerror(int errnum);
size_t strlen(const char *s);
char * strdup(const char *s);
char * strndup(const char *s, size_t size);

Only if the implementation defines __STDC_LIB_EXT1__ and additionally the user code defines __STDC_WANT_LIB_EXT1__ before any inclusion of <string.h>:

errno_t memcpy_s(void * restrict s1, rsize_t s1max, const void * restrict s2, rsize_t n);
errno_t memmove_s(void *s1, rsize_t s1max, const void *s2, rsize_t n);
errno_t strcpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2, size_t n);
errno_t strncpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2, rsize_t n);
errno_t strcat_s(char * restrict s1, rsize_t s1max, const char * restrict s2, size_t n);
errno_t strncat_s(char * restrict s1, rsize_t s1max, const char * restrict s2, rsize_t n);
char * strtok_s(char * restrict s1, rsize_t * restrict s1max, const char * restrict s2, char ** restrict ptr);
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n);
errno_t strerror_s(char *s, rsize_t maxsize, errno_t errnum);
size_t strerrorlen_s(errno_t errnum);
size_t strnlen_s(const char *s, size_t maxsize);

B.24 Type-generic math <tgmath.h>

acos acospi fabs exp2 lgamma nexttoward trunc
asin asinpi expm1 llogb nextup ufromfp
atan atandpi fdim llrint pown ufromfp
acos atancos hypot remainder u8sqrt
asin asinhy sinh
atan atanh cos cos2f
atan2 atanh cos2f
floor llround powr fadd
asin1 fmaxmag log10p1remainder dadd
cosh acosh log10 remquo fsub
sin sinh lrint dind
sinh sinh log10p1 roothn fmul
tanh atanh log2 roundr copysign
compound ln log2 round even fmod
exp exp10f logb droundf
cosh cosp sign sin
copysign mod logb round ddiv
cosh cosp fromfp logp1 rsoft ddiv
tanh tg hypot nearbyint sinpi fsgnt
exp erf fromfpm lrint cbln fma
exp10 expm1 hypot nearbyint sinpi fsgnt
pow exp10 ilogb nextafter tanpi dsgnt
sqrt exp2f ldexp nextdown tgamma

Only if the implementation does not define __STDC_NO_COMPLEX__: 
Only if the implementation defines \texttt{__STDC\_IEC\_60559\_DFP__}:

\begin{tabular}{cccccc}
\texttt{d32add} & \texttt{d64sub} & \texttt{d32div} & \texttt{d64fma} & \texttt{quantize} & \texttt{llquantexp} \\
\texttt{d64add} & \texttt{d32mul} & \texttt{d64div} & \texttt{d32sqrt} & \texttt{samequantum} & \\
\texttt{d32sub} & \texttt{d64mul} & \texttt{d32fma} & \texttt{d64sqrt} & \texttt{quantum} \\
\end{tabular}

\section*{B.25 Threads <threads.h>}

\begin{tabular}{lll}
\texttt{\_\_STDC\_NO\_THREADS\_} & \texttt{mtx\_t} & \texttt{thrd\_timedout} \\
\texttt{thread\_local} & \texttt{tss\_dtor\_t} & \texttt{thrd\_success} \\
\texttt{ONCE\_FLAG\_INIT} & \texttt{thrd\_start\_t} & \texttt{thrd\_busy} \\
\texttt{TSS\_DTOR\_ITERATIONS} & \texttt{once\_flag} & \texttt{thrd\_error} \\
\texttt{cnd\_t} & \texttt{mtx\_plain} & \texttt{thrd\_nomem} \\
\texttt{thrd\_t} & \texttt{mtx\_recursive} & \\
\texttt{tss\_t} & \texttt{mtx\_timed} & \\
\end{tabular}

\begin{verbatim}
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
   const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
_Noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
void thrd_yield(void);
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
\end{verbatim}

\section*{B.26 Date and time <time.h>}

\begin{tabular}{lll}
\texttt{NULL} & \texttt{size\_t} & \texttt{struct timespec} \\
\texttt{CLOCKS\_PER\_SEC} & \texttt{clock\_t} & \texttt{struct tm} \\
\texttt{TIME\_UTC} & \texttt{time\_t} & \\
\end{tabular}

\begin{verbatim}
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
int timespec_get(struct timespec *ts, int base);
\end{verbatim}
char *asctime(const struct tm *timeptr);
char *asctime_r(const struct tm *timeptr, char *buf);
char *ctime(const time_t *timer);
char *ctime_r(const time_t *timer, char *buf);
struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *timer, struct tm *buf);
struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *timer, struct tm *buf);
size_t strftime(char *restrict s, size_t maxsize, const char *restrict format,
                 const struct tm *restrict timeptr);

errno_t rsize_t
errno_t asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);
errno_t ctime_s(char *s, rsize_t maxsize, const time_t *timer);
struct tm *gmtime_s(const time_t *timer, struct tm *result);
struct tm *localtime_s(const time_t *timer, struct tm *result);

B.27 Unicode utilities <uchar.h>

mbstate_t size_t char16_t char32_t

size_t mbtowcs(char16_t * restrict pc16, const char * restrict s, size_t n,
                mbstate_t * restrict ps);
size_t wctomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
size_t mbtowcs(char32_t * restrict pc32, const char * restrict s, size_t n,
                mbstate_t * restrict ps);
size_t wcrtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);

B.28 Extended multibyte/wide character utilities <wchar.h>

wchar_t size_t wint_t
mbstate_t struct tm

wchar_t * restrict s,
size_t n,
const wchar_t * restrict format,
va_list arg);
int vfwscanf(const wchar_t * restrict s, const wchar_t * restrict format,
             va_list arg);
int vfswscanf(FILE * restrict stream, const wchar_t * restrict format,
              va_list arg);
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
int fwpprintf(FILE * restrict stream, const wchar_t * restrict format, ...
             va_list arg);
int fvwprintf(FILE * restrict stream, const wchar_t * restrict format, ...
             va_list arg);
int vswscanf(FILE * restrict stream, const wchar_t * restrict format,
             va_list arg);
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format,
              va_list arg);
int vwscanf(const wchar_t * restrict s, const wchar_t * restrict format,
            va_list arg);
int vwprintf(const wchar_t * restrict format, ...);
int vwscanf(const wchar_t * restrict format, va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
int wcscntc(FILE *stream);
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream)</code></td>
<td>Get wide Char String from File</td>
</tr>
<tr>
<td><code>wint_t fputwc(wchar_t c, FILE *stream)</code></td>
<td>Put Wide Character to Stream</td>
</tr>
<tr>
<td><code>int fputws(const wchar_t * restrict s, FILE * restrict stream)</code></td>
<td>Put Wide String to Stream</td>
</tr>
<tr>
<td><code>int fwide(FILE *stream, int mode)</code></td>
<td>Set Full Wide Mode on Stream</td>
</tr>
<tr>
<td><code>wint_t getwc(FILE *stream)</code></td>
<td>Get Single Wide Character from Stream</td>
</tr>
<tr>
<td><code>wint_t getwchar(void)</code></td>
<td>Get Single Wide Character from Stream</td>
</tr>
<tr>
<td><code>wint_t putwc(wchar_t c, FILE *stream)</code></td>
<td>Put Single Wide Character to Stream</td>
</tr>
<tr>
<td><code>wint_t putwchar(wchar_t c)</code></td>
<td>Put Single Wide Character to Stream</td>
</tr>
<tr>
<td><code>wint_t ungetwc(wint_t c, FILE *stream)</code></td>
<td>Unget Single Wide Character from Stream</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr)</code></td>
<td>Convert Wide Char String to Double</td>
</tr>
<tr>
<td><code>float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr)</code></td>
<td>Convert Wide Char String to Float</td>
</tr>
<tr>
<td><code>long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr)</code></td>
<td>Convert Wide Char String to Long Double</td>
</tr>
<tr>
<td><code>long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)</code></td>
<td>Convert Wide Char String to Long Integer</td>
</tr>
<tr>
<td><code>int wcscmp(const wchar_t *s1, const wchar_t *s2)</code></td>
<td>Compare Wide Char Strings</td>
</tr>
<tr>
<td><code>int wcscoll(const wchar_t *s1, const wchar_t *s2)</code></td>
<td>Compare Wide Char Strings, Collating</td>
</tr>
<tr>
<td><code>int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n)</code></td>
<td>Compare Wide Char Strings, N-Bytes</td>
</tr>
<tr>
<td><code>int wcscmpncpy(const wchar_t *s1, const wchar_t *s2)</code></td>
<td>Compare Wide Char Strings, Copy N-Bytes</td>
</tr>
<tr>
<td><code>size_t wcsxfrm(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n)</code></td>
<td>Convert Wide Char String to Unicode Representation</td>
</tr>
<tr>
<td><code>wchar_t *wcscat(const wchar_t * restrict s, const wchar_t * restrict s2)</code></td>
<td>Concatenate Wide Char Strings</td>
</tr>
<tr>
<td><code>wchar_t *wcsncat(const wchar_t * restrict s, const wchar_t * restrict s2, size_t n)</code></td>
<td>Concatenate Wide Char Strings, N-Bytes</td>
</tr>
<tr>
<td><code>wchar_t *wcsncpy(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n)</code></td>
<td>Copy Wide Char Strings, N-Bytes</td>
</tr>
<tr>
<td><code>wchar_t *wcscspn(const wchar_t * restrict s, const wchar_t * restrict c)</code></td>
<td>Skip Wide Char Strings, N-Bytes</td>
</tr>
<tr>
<td><code>wchar_t *wcsxfrm(const wchar_t * restrict s1, const wchar_t * restrict s2, size_t n)</code></td>
<td>Convert Wide Char String to Unicode Representation</td>
</tr>
<tr>
<td><code>wchar_t *wcscnvn(const wchar_t * restrict s1, const wchar_t * restrict s2)</code></td>
<td>Convert Wide Char String to Unicode Representation</td>
</tr>
<tr>
<td><code>wchar_t *wcschr(const wchar_t * restrict s)</code></td>
<td>Get First Wide Character in String</td>
</tr>
<tr>
<td><code>wchar_t *wcsrchr(const wchar_t * restrict s)</code></td>
<td>Get Last Wide Character in String</td>
</tr>
<tr>
<td><code>size_t wcscspn(const wchar_t * restrict s1, const wchar_t * restrict c)</code></td>
<td>Skip Wide Char Strings, N-Bytes</td>
</tr>
<tr>
<td><code>wchar_t *wcscstr(const wchar_t * restrict s)</code></td>
<td>Get Last Wide Character in String</td>
</tr>
<tr>
<td><code>wchar_t *wcscstr(const wchar_t * restrict s)</code></td>
<td>Get Last Wide Character in String</td>
</tr>
<tr>
<td><code>size_t wcsftime(const wchar_t * restrict s, size_t n, const wchar_t * restrict format, const struct tm * restrict tm)</code></td>
<td>Convert Wide Char String to Time Representation</td>
</tr>
<tr>
<td><code>wint_t btwocw(int c)</code></td>
<td>Convert Wide Character to Text</td>
</tr>
<tr>
<td><code>int wctob(wint_t c)</code></td>
<td>Convert Text Character to Wide Character</td>
</tr>
<tr>
<td><code>int mbinit(const mbstate_t *ps)</code></td>
<td>Initialize Multi-Byte State</td>
</tr>
<tr>
<td><code>size_t mbtlen(const char * restrict s, size_t n, mbstate_t * restrict ps)</code></td>
<td>Get Multi-Byte Length</td>
</tr>
<tr>
<td><code>size_t mbtowc(const wchar_t * restrict wc, const char * restrict s, size_t n, mbstate_t * restrict ps)</code></td>
<td>Convert Multi-Byte to Wide Character</td>
</tr>
<tr>
<td><code>size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps)</code></td>
<td>Convert Wide Character to Multi-Byte</td>
</tr>
<tr>
<td><code>size_t mbstowcs(const wchar_t * restrict s, const char * restrict src, size_t n, mbstate_t * restrict ps)</code></td>
<td>Convert Wide Character to Multi-Byte String</td>
</tr>
<tr>
<td><code>size_t wcstombs(const char * restrict s, const wchar_t * restrict src, size_t n, mbstate_t * restrict ps)</code></td>
<td>Convert Wide Character to Multi-Byte String</td>
</tr>
</tbody>
</table>

Only if the implementation defines `__STDC_LIB_EXT1__` and additionally the user code defines `__STDC_WANT_LIB_EXT1__` before any inclusion of `<wchar.h>`.
errno_t  rsize_t

int fwprintf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
int swprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, ...);
int sswprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, va_list arg);
int vfwprintf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vfwscanf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
int vsnwprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, va_list arg);
int vswprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, va_list arg);
int vswscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
int vwprintf_s(const wchar_t * restrict format, va_list arg);
int vwscanf_s(const wchar_t * restrict format, va_list arg);

errno_t wcscpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
errno_t wcsncpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
errno_t wmemcpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
errno_t wcscat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
errno_t wcsncat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);

wchar_t * wcstok_s(wchar_t * restrict s1, rsize_t * restrict s1max, const wchar_t * restrict s2, wchar_t ** restrict ptr);
size_t wcsnlen_s(const wchar_t * restrict s, size_t maxsize);
errno_t wcrtomb_s(size_t * restrict retval, char * restrict s, size_t smax, wchar_t wc, mbstate_t * restrict ps);
errno_t mbsrtowcs_s(size_t * restrict retval, wchar_t * restrict dst, rsize_t dstmax, const char ** restrict src, rsize_t len, mbstate_t * restrict ps);
errno_t wcsrtombs_s(size_t * restrict retval, char * restrict dst, rsize_t dstmax, const wchar_t ** restrict src, rsize_t len, mbstate_t * restrict ps);

B.29 Wide character classification and mapping utilities <wctype.h>

wint_t  wctrans_t  wctype_t  WEOF

int iswalnum(wint_t wc);
inwAlpha(wint_t wc);
inwBlank(wint_t wc);
inwCtrl(wint_t wc);
inwDigit(wint_t wc);
inwGraph(wint_t wc);
inwLower(wint_t wc);
inwPrint(wint_t wc);
inwPunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char *property);
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
wint_t towctrans(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char *property);
Annex C
(informative)

Sequence points

The following are the sequence points described in 5.1.2.3:

— Between the evaluations of the function designator and actual arguments in a function call and the actual call. (6.5.2.2).

— Between the evaluations of the first and second operands of the following operators: logical AND && (6.5.13); logical OR || (6.5.14); comma , (6.5.17).

— Between the evaluations of the first operand of the conditional ?: operator and whichever of the second and third operands is evaluated (6.5.15).

— Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions: a full declarator for a variably modified type; an initializer that is not part of a compound literal (6.7.9); the expression in an expression statement (6.8.3); the controlling expression of a selection statement (if or switch) (6.8.4); the controlling expression of a while or do statement (6.8.5); each of the (optional) expressions of a for statement (6.8.5.3); the (optional) expression in a return statement (6.8.6.4).

— Immediately before a library function returns (7.1.4).

— After the actions associated with each formatted input/output function conversion specifier (7.21.6, 7.29.2).

— Immediately before and immediately after each call to a comparison function, and also between any call to a comparison function and any movement of the objects passed as arguments to that call (7.22.5).
Annex D  
Universal character names for identifiers

This clause lists the hexadecimal code values that are valid in universal character names in identifiers.

D.1 Ranges of characters allowed

1. 00A8, 00AA, 00AD, 00AF, 00B2–00B5, 00B7–00BA, 00BC–00BE, 00C0–00D6, 00D8–00F6, 00F8–00FF
2. 0100–167F, 1681–180D, 180F–1FFF
4. 2070–218F, 2460–24FF, 2776–2793, 2C00–2DFF, 2E80–2FFF
5. 3004–3007, 3021–302F, 3031–303F
6. 3040–D7FF

D.2 Ranges of characters disallowed initially

1. 0300–036F, 1DC0–1DFF, 20D0–20FF, FE20–FE2F
Annex E  
(informative)  
Implementation limits

The contents of the header `<limits.h>` are given below, in alphabetical order. The minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign. The values shall all be constant expressions suitable for use in `#if` preprocessing directives. The components are described further in 5.2.4.2.1.

```
#define CHAR_BIT 8
#define CHAR_MAX UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN 0 or SCHAR_MIN
#define INT_MAX +32767
#define INT_MIN -32767
#define LONG_MAX +2147483647
#define LONG_MIN -2147483647
#define LONGLONG_MAX +9223372036854775807
#define LONGLONG_MIN -9223372036854775807
#define MB_LEN_MAX 1
#define SCHAR_MAX +127
#define SCHAR_MIN -127
#define SHRT_MAX +32767
#define SHRT_MIN -32767
#define UCHAR_MAX 255
#define USHRT_MAX 65535
#define UINT_MAX 65535
#define ULONG_MAX 4294967295
#define ULLONG_MAX 18446744073709551615
```

The contents of the header `<float.h>` are given below. All integer values, except `FLT_ROUNDS`, shall be constant expressions suitable for use in `#if` preprocessing directives; all floating values shall be constant expressions. The components are described further in 5.2.4.2.2.

```
#define FLT_EVAL_METHOD
#define FLT_ROUNDS
ifdef __STDC_IEC_60559_DFP__
#define DEC_EVAL_METHOD
endif
```

The values given in the following list shall be replaced by implementation-defined expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

```
#define DBL_DECIMAL_DIG 10
#define DBL_DIG 10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP -37
#define DBL_MIN_EXP
#define DECIMAL_DIG 10
#define DECIMAL_DIG
#define FLT_DECIMAL_DIG 6
#define FLT_DIG 6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP -37
#define FLT_MIN_EXP
```
5 The values given in the following list shall be replaced by implementation-defined constant expressions with values that are greater than or equal to those shown:

```
#define DBL_MAX 1E+37
#define DBL_NORM_MAX 1E+37
#define FLT_MAX 1E+37
#define FLT_NORM_MAX 1E+37
#define LDBL_MAX 1E+37
#define LDBL_NORM_MAX 1E+37
```

6 The values given in the following list shall be replaced by implementation-defined constant expressions with (positive) values that are less than or equal to those shown:

```
#define DBL_EPSILON 1E-9
#define DBL_MIN 1E-37
#define FLT_EPSILON 1E-5
#define FLT_MIN 1E-37
#define LDBL_EPSILON 1E-9
#define LDBL_MIN 1E-37
```

7 If the implementation supports decimal floating types, the following macros provide the parameters of these types as exact values.

```
#ifndef __STDC_IEC_60559_DFP__
#define DEC32_EPSILON 1E-6DF
#define DEC32_MANT_DIG 7
#define DEC32_MAX 9.999999E96DF
#define DEC32_MAX_EXP 97
#define DEC32_MIN 1E-95DF
#define DEC32_MIN_EXP -94
#define DEC32_TRUE_MIN 0.000001E-95DF
#define DEC64_EPSILON 1E-15DD
#define DEC64_MANT_DIG 16
#define DEC64_MAX 9.9999999999999999E384DD
#define DEC64_MAX_EXP 385
#define DEC64_MIN 1E-383DD
#define DEC64_MIN_EXP -382
#define DEC64_TRUE_MIN 0.000000000000001E-383DD
#define DEC128_EPSILON 1E-33DL
#define DEC128_MANT_DIG 34
#define DEC128_MAX 9.99999999999999999999999999999999E6144DL
#define DEC128_MAX_EXP 6145
#define DEC128_MIN 1E-6143DL
#define DEC128_MIN_EXP -6142
#define DEC128_TRUE_MIN 0.000000000000000000000000000000001E-6143DL
#endif
```
Annex F
(normative)
IEC 60559 floating-point arithmetic

F.1 Introduction


3 An implementation that defines __STDC_IEC_559__ __STDC_IEC_60559_BFP__ to yyyyml shall conform to the specifications in this annex for binary floating-point arithmetic and shall also define __STDC_IEC_559__ to 1.\(^{379}\)

4 An implementation that defines __STDC_IEC_60559_DFP__ to yyyyml shall conform to the specifications for decimal floating-point arithmetic in the following subclauses of this annex:

| F.2.1 | Infinities and NaNs |
| F.3 | Operations |
| F.4 | Floating to integer conversions |
| F.6 | The return statement |
| F.7 | Contracted expressions |
| F.8 | Floating-point environment |
| F.9 | Optimization |
| F.10 | Mathematics <math.h> and <tgmath.h> |

For the purpose of specifying these conformance requirements, the macros, functions, and values mentioned in the subclauses listed above are understood to refer to the corresponding macros, functions, and values for decimal floating types. Likewise, the “rounding direction mode” is understood to refer to the rounding direction mode for decimal floating-point arithmetic.

5 Where a binding between the C language and IEC 6059 is indicated, the IEC 60559-specified behavior is adopted by reference, unless explicitly stated otherwise. Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values.

6 This annex amends some standard headers with declarations or definitions of identifiers contingent on whether certain macros whose names begin with __STDC_WANT_IEC_60559__ and end with __EXT__ are defined (by the user) at the point in the code where the header is first included. Within a preprocessing translation unit, the same set of such macros shall be defined for the first inclusion of all such headers.

\(^{379}\)Implementations that do not define either of __STDC_IEC_60559_BFP__ and __STDC_IEC_559__ are not required to conform to these specifications. New code should not use the obsolescent macro __STDC_IEC_559__ to test for conformance to this annex.
### F2 Types

1. The C floating types match the IEC 60559 formats as follows:
   - The `float` type matches the IEC 60559 single-binary32 format.
   - The `double` type matches the IEC 60559 double-binary64 format.
   - The `long double` type matches the IEC 60559 binary128 format, else an IEC 60559 extended binary64-extended format, else a non-IEC 60559 extended format, else the IEC 60559 double-binary64 format.

Any non-IEC 60559 extended format used for the `long double` type shall have more precision than IEC 60559 double-binary64 and at least the range of IEC 60559 double-binary64. The value of `FLT_ROUNDS` applies to all IEC 60559 types supported by the implementation, but need not apply to non-IEC 60559 types.

**Recommended practice**

2. The `long double` type should match the IEC 60559 binary128 format, else an IEC 60559 extended format, binary64-extended format.

### F2.1 Infinities and NaNs

1. This specification does not define the behavior of signaling NaNs. Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values (5.2.4.2.2).

2. It generally uses the term NaN to denote quiet NaNs. The `NAN` and `INFINITY` macros and the `isnan`, `nan` functions in `<math.h>` provide designations for IEC 60559 quiet NaNs and infinities.

3. C operators and functions provide `SNANF`, `SNAN`, and `SNANL` macros in `<math.h>` provide designations for IEC 60559 required and recommended facilities as listed below. The `+`, `-`, `*`, and `/` signaling NaNs.

4. Any operator or `<math.h>` provide the function that raises an “invalid” floating-point exception, if delivering a floating type result, shall return a quiet NaN.

5. In order to support signaling NaNs as specified in IEC 60559, an operation that rounds a floating-point number to an integer value (in the same precision). The `nearbyint`, an implementation should adhere to the following recommended practice.

**Recommended practice**

6. Functions in `<math.h>` provide the `nearbyint` function recommended in the Appendix to ANSI/IEEE 854. The conversions for floating types provide the IEC 60559 conversions between function or macro with a signaling NaN input, unless explicitly specified otherwise, raises an “invalid” floating-point exceptions. The conversions from integer to floating types provide the IEC 60559 conversions from integer to floating point. The conversions

---

380) “Extended” is IEC 60559 4 double-extended data format. Extended refers to both binary64-extended formats include the common 80-bit and quadruple-128 bit IEC 60559 formats format.

381) A non-IEC 60559 long double type is required to provide infinity and NaNs, as its values include all double values.

382) Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.
from floating to integer types provide IEC 60559 like conversions but always round toward zero. The lrint exception.

7 NOTE Some functions do not propagate quiet NaN arguments. For example, hypot(x, y) returns infinity if x or y is infinite and the other is a quiet NaN. The recommended practice in this subclause specifies that such functions (and others) raise the "invalid" floating-point exception if an argument is a signaling NaN, which also implies they return a quiet NaN in these cases.

8 and llrint The <fenv.h> header defines the macro FE_SNANS_ALWAYS_SIGNAL if and only if the implementation follows the recommended practice in this subclause. If defined, FE_SNANS_ALWAYS_SIGNAL expands to the integer constant 1.

E.3 Operations

1 functions in provide the C operators, functions, and function-like macros provide the operations required by IEC 60559 conversions, which honor the directed rounding mode, from floating point to the long int and long long int integer formats. The lrint and llrint functions can be used to implement as shown in the following table. Specifications for the C facilities are provided in the listed clauses. The C specifications are intended to match IEC 60559 conversions from floating to other integer formats. The translation time conversion of floating constants and the +, -, *, /, and related library functions in and provide, unless stated otherwise.

<table>
<thead>
<tr>
<th>IEC 60559 operation</th>
<th>C operation</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>roundToIntegralTiesToEven</td>
<td>roundeven</td>
<td>7.12.9.8, F.10.6.8</td>
</tr>
<tr>
<td>roundToIntegralTiesAway_</td>
<td>round</td>
<td>7.12.9.6, F.10.6.6</td>
</tr>
<tr>
<td>roundToIntegralTowardZero_</td>
<td>trunc</td>
<td>7.12.9.9, F.10.6.9</td>
</tr>
<tr>
<td>roundToIntegralTowardPositive</td>
<td>ceil</td>
<td>7.12.9.1, F.10.6.1</td>
</tr>
<tr>
<td>roundToIntegralTowardNegative</td>
<td>floor</td>
<td>7.12.9.2, F.10.6.2</td>
</tr>
<tr>
<td>roundToIntegralExact</td>
<td>rint</td>
<td>7.12.9.4, F.10.6.4</td>
</tr>
<tr>
<td>nextUp</td>
<td>nextup</td>
<td>7.12.11.5, F.10.8.5</td>
</tr>
<tr>
<td>nextDown</td>
<td>nextdown</td>
<td>7.12.11.6, F.10.8.6</td>
</tr>
<tr>
<td>remainder</td>
<td>remainder, remquo</td>
<td>7.12.10.2, F.10.7.2,</td>
</tr>
<tr>
<td>minNum</td>
<td>min</td>
<td>7.12.12.3, F.10.9.3</td>
</tr>
<tr>
<td>maxNum</td>
<td>max</td>
<td>7.12.12.2, F.10.9.2</td>
</tr>
<tr>
<td>minNumMag</td>
<td>fmin</td>
<td>7.12.12.5, F.10.9.5</td>
</tr>
<tr>
<td>maxNumMag</td>
<td>fmax</td>
<td>7.12.12.4, F.10.9.4</td>
</tr>
<tr>
<td>scaleB</td>
<td>scalbn, scalbln</td>
<td>7.12.6.19, F.10.3.19</td>
</tr>
<tr>
<td>logB</td>
<td>logb, ilogb, llogb</td>
<td>7.12.6.17, F.10.3.17,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.12.6.8, F.10.3.8,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.12.6.10, F.10.3.10</td>
</tr>
<tr>
<td>addition</td>
<td>+, fadd, faddl, daddl</td>
<td>6.5.6, 7.12.14.1, F.10.11</td>
</tr>
<tr>
<td>subtraction</td>
<td>-, fsub, fsubl, dsbl</td>
<td>6.5.6, 7.12.14.2, F.10.11</td>
</tr>
<tr>
<td>multiplication</td>
<td>*, fmul, fmulll, dmulll</td>
<td>6.5.5, 7.12.14.3, F.10.11</td>
</tr>
<tr>
<td>division</td>
<td>/, fdiv, fdivl, ddivl</td>
<td>6.5.5, 7.12.14.4, F.10.11</td>
</tr>
<tr>
<td>squareRoot</td>
<td>sqrt, fsqrt, fsqrtl, dsqrtl</td>
<td>7.12.7.10, F.10.4.10,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.12.14.6, F.10.11</td>
</tr>
<tr>
<td>fusedMultiplyAdd</td>
<td>fma, ffma, ffmal, dfmal</td>
<td>7.12.13.1, F.10.10.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.12.14.5, F.10.11</td>
</tr>
</tbody>
</table>
| convertFromInt                    | cast and implicit conversion | 6.3.1.4, 6.5.4
<table>
<thead>
<tr>
<th>Function Description</th>
<th>Function Call</th>
<th>Page References</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvertToIntegerTiesToEven</td>
<td>fromfp, ufromfp</td>
<td>7.12.9.10, F.10.6.10</td>
</tr>
<tr>
<td>ConvertToIntegerTowardZero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertToIntegerTowardPositive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertToIntegerTiesToAway</td>
<td>fromfp, ufromfp, lround, lround</td>
<td>7.12.9.10, F.10.6.10, 7.12.9.7, F.10.6.7</td>
</tr>
<tr>
<td>ConvertToIntegerExactTiesToEven</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertToIntegerExactTowardZero</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertToIntegerExactTowardPositive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertToIntegerExactTiesToAway</td>
<td>fromfp, ufromfp</td>
<td>7.12.9.11, F.10.6.11</td>
</tr>
<tr>
<td>ConvertFormat - different formats</td>
<td>cast and implicit conversions</td>
<td>6.3.1.5, 6.5.4</td>
</tr>
<tr>
<td>ConvertFormat - same format</td>
<td>canonicalize</td>
<td>7.12.11.7, F.10.8.7</td>
</tr>
<tr>
<td>ConvertFromDecimalCharacter</td>
<td>strtod, wcstod, scanf, wscanf, decimal floating constants</td>
<td>7.22.1.5, 7.29.1.1, 7.21.6.4, 7.29.2.12, E5</td>
</tr>
<tr>
<td>ConvertToDecimalCharacter</td>
<td>printf, wprintf, strftime</td>
<td>7.21.6.3, 7.29.2.11, 7.22.1.3, F.5</td>
</tr>
<tr>
<td>ConvertFromHexCharacter</td>
<td>strtod, wcstod, scanf, wscanf, hexadecimal floating constants</td>
<td>7.22.1.5, 7.29.1.1, 7.21.6.4, 7.29.2.12, E5</td>
</tr>
<tr>
<td>ConvertToHexCharacter</td>
<td>printf, wprintf, strftime</td>
<td>7.21.6.3, 7.29.2.11, 7.22.1.3, F.5</td>
</tr>
<tr>
<td>Copy</td>
<td>memcpy, memmove</td>
<td>7.24.2.1, 7.24.2.3</td>
</tr>
<tr>
<td>Negate</td>
<td>-(x)</td>
<td>6.5.3.3</td>
</tr>
<tr>
<td>Abs</td>
<td>fabs</td>
<td>7.12.7.3, F.10.4.3</td>
</tr>
<tr>
<td>CopySign</td>
<td>copysign</td>
<td>7.12.11.1, F.10.8.1</td>
</tr>
<tr>
<td>CompareQuietEqual</td>
<td>==</td>
<td>6.5.9, F.9.3</td>
</tr>
<tr>
<td>CompareQuietNotEqual</td>
<td>!=</td>
<td>6.5.9, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingEqual</td>
<td>iseqsig</td>
<td>7.12.17.7, F.10.14.1</td>
</tr>
<tr>
<td>CompareSignalingGreater</td>
<td>&gt;</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingGreaterEqual</td>
<td>&gt;=</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingLess</td>
<td>&lt;</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingLessEqual</td>
<td>&lt;=</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingNotEqual</td>
<td>! iseqsig(x)</td>
<td>7.12.17.7, F.10.14.1</td>
</tr>
<tr>
<td>CompareSignalingNotGreater</td>
<td>! (x &gt; y)</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingNotLess</td>
<td>! (x &lt; y)</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingLessUnordered</td>
<td>! (x &lt;= y)</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareSignalingNotGreaterUnordered</td>
<td>! (x &gt;= y)</td>
<td>6.5.8, F.9.3</td>
</tr>
<tr>
<td>CompareQuietGreater</td>
<td>isgreater</td>
<td>7.12.17.1</td>
</tr>
<tr>
<td>CompareQuietGreaterEqual</td>
<td>isgreateerequal</td>
<td>7.12.17.2</td>
</tr>
<tr>
<td>CompareQuietLess</td>
<td>isless</td>
<td>7.12.17.3</td>
</tr>
<tr>
<td>CompareQuietLessEqual</td>
<td>islessequal</td>
<td>7.12.17.4</td>
</tr>
<tr>
<td>CompareQuietUnordered</td>
<td>isunordered</td>
<td>7.12.17.6</td>
</tr>
<tr>
<td>CompareQuietNotGreater</td>
<td>! isgreater(x, y)</td>
<td>7.12.17.1</td>
</tr>
<tr>
<td>CompareQuietLessUnordered</td>
<td>! isgreateerequal(x, y)</td>
<td>7.12.17.2</td>
</tr>
<tr>
<td>CompareQuietNotLess</td>
<td>! isless(x, y)</td>
<td>7.12.17.3</td>
</tr>
<tr>
<td>CompareQuietGreaterUnordered</td>
<td>! islessequal(x, y)</td>
<td>7.12.17.4</td>
</tr>
<tr>
<td>CompareQuietOrdered</td>
<td>! isunordered(x, y)</td>
<td>7.12.17.6</td>
</tr>
<tr>
<td>Class</td>
<td>fpclassify, signbit, isnormal, isfinite</td>
<td>7.12.3.1, 7.12.3.7, 7.12.3.8</td>
</tr>
<tr>
<td>IsSignMinus</td>
<td>signbit</td>
<td>7.12.3.7</td>
</tr>
<tr>
<td>IsNormal</td>
<td>isnormal</td>
<td>7.12.3.6</td>
</tr>
<tr>
<td>IsFinite</td>
<td>isfinite</td>
<td>7.12.3.3</td>
</tr>
</tbody>
</table>
The IEC 60559 binary floating-point conversions. The function in provides the `conv` function recommended in the Appendix to ANSI/IEEE 854. The relational and equality operators provide requirement that certain of its operations be provided for operands of different formats (of the same radix) is satisfied by C's usual arithmetic conversions (6.3.1.8) and function-call argument conversions (6.5.2.2). For example, the following operations take `float` \( f \) and `double` \( d \) inputs and produce a `long double` result:

\[
    (\text{long double}) f + d \\
    \text{powl}(f, d)
\]

Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 conversion, IEC convertFormat or copy operation\(^{33} \) is implementation-defined, even if `<cenv.h>` defines the macro `FE_SNANS_ALWAYS_SIGNAL` (F.2.1). If the return expression of a `return` statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression.

The unary `-` operator raises no floating-point exceptions, even if the operand is a signaling NaN.

The C classification macros `fpclassify`, `iscanonical`, `isnan`, `isnormal`, `isnormal`, and `iszero` provide the IEC 60559 identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison macros (`<`, `<=`, `==`, and `!=`) supplement the language operators to address this need. The `islessgreater` and `isunordered` macros provide respectively a quiet version of the `<` predicate and the unordered predicate operations indicated in the table above provided their arguments are in the format of their semantic type. Then these macros raise no floating-point exceptions, even if an argument is a signaling NaN.

The C `nearbyint` functions (7.12.9.3, F.10.6.3) provide the `nearbyint` function recommended in the Appendix to IEC 60559. The `...` and `...` functions in provide the facility to test and alter the (superseded) ANSI/IEEE 854.

The C `nextafter` (7.12.11.3, F.10.8.3) and `nexttoward` (7.12.11.4, F.10.8.4) functions provide

\(^{33}\)Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the "invalid" floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.
the nextafter function recommended in the Appendix to (superseded) IEC 60559 floating-point exception status flags. The and functions in provide the facility to save and restore all five status flags at once. These functions are used in conjunction with the type -1989 (but with a minor change to better handle signed zeros).

8 The C getpayload_setpayload, and setpayloadsig (F:10.13) functions provide program access to NaN payloads, defined in IEC 60559.

9 The macros (7.6) FE_DOWNWARD, FE_TONEAREST, FE_TONEARESTFROMZERO, FE_TOWARDZERO, and FE_UPWARD, which are used in conjunction with the fegetround and fesetround functions and the floating-point exception macros (,..., ) also in ... The and functions in provide the facility to select among FENV_ROUND pragma, represent the IEC 60559 directed rounding modes represented by the rounding direction macros in (...,...) and the values 0, 1, 2, and 3 of are the IEC 60559 directed rounding modes. The rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTowardZero, and functions in roundTowardPositive, respectively. Support for the roundTiesToAway attribute for binary floating-point arithmetic, and hence for the FE_TONEARESTFROMZERO macro, is optional.

10 The C fegetenv (7.6.6.1), feholdexcept (7.6.6.2), fesetenv (7.6.6.3) and feupdateenv (7.6.6.4) functions provide a facility to manage the dynamic floating-point environment, comprising the IEC 60559 status flags and dynamic control modes. The copySign.

11 functions in provide the copySign function recommended in the Appendix IEC 60559 requires operations with specified operand and result formats. Therefore, math functions that are bound to IEC 60559. The fabs 60559 operations (see table above) must remove any extra range and precision from arguments or results.

12 functions in provide the abs function recommended in the Appendix to IEC 60559. The unary minus (−) 60559 requires operations that round their result to formats the same as and wider than the operands, in addition to the operations that round their result to narrower formats (see 7.12.14). Operators (−, −, *, and /) whose evaluation formats are wider than the semantic type (5.2.4.2.2) might not support some of the IEC 60559 operations, because getting a result in a given format might require a cast that could introduce an extra rounding error. The functions that round result to narrower type (7.12.14) provide the IEC 60559 operations that round result to same and wider (as well as narrower) formats, in those cases where built-in operators and casts do not. For example, dddivl(x, y) computes a correctly rounded double divide of float x by float y, regardless of the evaluation method.

13 operator provides the unary minus (−) operation recommended in the Appendix to Decimal versions of the remquo library function are not provided. (The decimal remainder functions provide the remainder operation defined by IEC 60559.)

14 The and functions in provide the scalb function recommended in the Appendix to quantizedN functions (7.12.15.1) provide the quantize operation defined in IEC 60559 for decimal floating-point arithmetic.

15 The binding for the convertFormat operation applies to all conversions among IEC 60559 formats. Therefore, for implementations that conform to Annex F, conversions between decimal floating types and standard floating types with IEC 60559 formats are correctly rounded and raise floating-point exceptions as specified in IEC 60559. The Logb.

16 functions in provide the logb function recommended in the Appendix to IEC 60559 specifies the convertFromHexCharacter and convertToHexCharacter operations only for binary floating-point arithmetic.

17 The integer constant 18 provides the radix operation defined in IEC 60559, but following the newer specifications in ANSI/IEEE 854-1989 for decimal floating-point arithmetic.

The nextafter and nexttoward functions in provide the nextafter function recommended in the Appendix to The samequantumN functions (7.12.15.2) provide the sameQuantum operation defined in IEC 60559 (but with a minor change to better handle signed zeros). The macro provides the finite function recommended in the Appendix to IEC 60559. The macro in,...
floating-point arithmetic.

The `fe_dec_getround`(7.6.5.3) and `fe_dec_setround` (7.6.5.6) functions provide the getDecimalRoundingDirection and setDecimalRoundingDirection operations defined in IEC 60559 for decimal floating-point arithmetic. The macros (7.6) `FE_DEC_DOWNWARD`, `FE_DEC_TONEAREST`, `FE_DEC_TONEARESTFROMZERO`, `FE_DEC_TOWARDZERO`, and `FE_DEC_UPWARD` which are used in conjunction with the number classification macros (., .), provide the facility of the classification function recommended in the Appendix to `fe_dec_getround` and `fe_dec_setround` functions, represent the IEC 60559 rounding-direction attributes `roundTowardNegative`, `roundTiesToEven`, `roundTiesToAway`, `roundTowardZero`, and `roundTowardPositive`, respectively.

The quantum of N (7.12.15.3) and `llquantexpd`N (7.12.15.4) functions compute the quantum and the (quantum) exponent q defined in IEC 60559 for decimal numbers viewed as having integer significands.

The `encodedecn`N (7.12.16.1) and `decodedecn`N (7.12.16.2) functions provide the `encodeDecimal` and `decodeDecimal` operations defined in IEC 60559 (except that the classification macros defined in 7.12.3 do not distinguish signaling from quiet NaNs) for decimal floating-point arithmetic.

The `encodedbn`N (7.12.16.3) and `decodedbn`N (7.12.16.4) functions provide the `encodeBinary` and `decodeBinary` operations defined in IEC 60559 for decimal floating-point arithmetic.

The C functions in the following table provide operations recommended by IEC 60559 and similar operations. Correct rounding, which IEC 60559 specifies for its operations, is not required for the C functions in the table. See also 7.31.8.

<table>
<thead>
<tr>
<th>IEC 60559 operation</th>
<th>C function</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>exp</td>
<td>7.12.6.1, F.10.3.1</td>
</tr>
<tr>
<td>expm1</td>
<td>expm1</td>
<td>7.12.6.6, F.10.3.6</td>
</tr>
<tr>
<td>exp2</td>
<td>exp2</td>
<td>7.12.6.4, F.10.3.4</td>
</tr>
<tr>
<td>exp2m1</td>
<td>exp2m1</td>
<td>7.12.6.5, F.10.3.5</td>
</tr>
<tr>
<td>exp10</td>
<td>exp10</td>
<td>7.12.6.2, F.10.3.2</td>
</tr>
<tr>
<td>exp10m1</td>
<td>exp10m1</td>
<td>7.12.6.3, F.10.3.3</td>
</tr>
<tr>
<td>log</td>
<td>log</td>
<td>7.12.6.11, F.10.3.11</td>
</tr>
<tr>
<td>log2</td>
<td>log2</td>
<td>7.12.6.15, F.10.3.15</td>
</tr>
<tr>
<td>log10</td>
<td>log10</td>
<td>7.12.6.12, F.10.3.12</td>
</tr>
<tr>
<td>log1p, log1p</td>
<td>log1p, log1p</td>
<td>7.12.6.14, F.10.3.14</td>
</tr>
<tr>
<td>log2p1</td>
<td>log2p1</td>
<td>7.12.6.16, F.10.3.16</td>
</tr>
<tr>
<td>log10p1</td>
<td>log10p1</td>
<td>7.12.6.13, F.10.3.13</td>
</tr>
<tr>
<td>hypot</td>
<td>hypot</td>
<td>7.12.7.4, F.10.4.4</td>
</tr>
<tr>
<td>rSqrt</td>
<td>rSqrt</td>
<td>7.12.7.9, F.10.4.9</td>
</tr>
<tr>
<td>compound</td>
<td>compoundn</td>
<td>7.12.7.2, F.10.4.2</td>
</tr>
<tr>
<td>rootn</td>
<td>rootn</td>
<td>7.12.7.8, F.10.4.8</td>
</tr>
<tr>
<td>pown</td>
<td>pown</td>
<td>7.12.7.6, F.10.4.6</td>
</tr>
<tr>
<td>pow</td>
<td>pow</td>
<td>7.12.7.5, F.10.4.5</td>
</tr>
<tr>
<td>powr</td>
<td>powr</td>
<td>7.12.7.7, F.10.4.7</td>
</tr>
<tr>
<td>sin</td>
<td>sin</td>
<td>7.12.4.6, F.10.1.6</td>
</tr>
<tr>
<td>cos</td>
<td>cos</td>
<td>7.12.4.5, F.10.1.5</td>
</tr>
<tr>
<td>tan</td>
<td>tan</td>
<td>7.12.4.7, F.10.1.7</td>
</tr>
<tr>
<td>sinPi</td>
<td>sinPi</td>
<td>7.12.4.13, F.10.1.13</td>
</tr>
<tr>
<td>cosPi</td>
<td>cosPi</td>
<td>7.12.4.12, F.10.1.12</td>
</tr>
<tr>
<td>tanPi</td>
<td>tanPi</td>
<td>7.12.4.14, F.10.1.14</td>
</tr>
<tr>
<td>asinPi</td>
<td>asinPi</td>
<td>7.12.4.9, F.10.1.9</td>
</tr>
<tr>
<td>acosPi</td>
<td>acosPi</td>
<td>7.12.4.8, F.10.1.8</td>
</tr>
<tr>
<td>atanPi</td>
<td>atanPi</td>
<td>7.12.4.10, F.10.1.10</td>
</tr>
<tr>
<td>atan2Pi</td>
<td>atan2Pi</td>
<td>7.12.4.11, F.10.1.11</td>
</tr>
<tr>
<td>asin</td>
<td>asin</td>
<td>7.12.4.2, F.10.1.2</td>
</tr>
<tr>
<td>acos</td>
<td>acos</td>
<td>7.12.4.1, F.10.1.1</td>
</tr>
</tbody>
</table>

... continued ...
### F4 Floating to integer conversion

1. If the integer type is `_Bool`, and no floating-point exceptions are raised, the conversion raises no floating-point exceptions. Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the “invalid” floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by the following rules:
   - **Conversion of** `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, and `atanh` **functions**: `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, and `atanh`.
   - **Function to** decimal **digits**: `7.12.4.3, F.10.1.3`
   - **Function to** floating-point **digits**: `7.12.2.4, F.10.1.4`
   - **Function to** decimal **digits**: `7.12.5.5, F.10.2.5`
   - **Function to** floating-point **digits**: `7.12.5.4, F.10.2.4`
   - **Function to** decimal **digits**: `7.12.5.6, F.10.2.6`
   - **Function to** floating-point **digits**: `7.12.5.2, F.10.2.2`
   - **Function to** decimal **digits**: `7.12.5.1, F.10.2.1`
   - **Function to** floating-point **digits**: `7.12.5.3, F.10.2.3`

### F5 Conversions between binary floating types and decimal character sequences

1. **Conversion from the widest** floating-point **type** to **C** **function**?

   ```c
   CR_DECIMAL_DIG
   ```

   if and only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where `<float.h>` is first included. If defined, `CR_DECIMAL_DIG` expands to an integral constant expression suitable for use in `#if` preprocessing directives whose value is a number such that conversions between all supported IEC 60559 format to decimal with digits and back is the identity function. For binary formats and character sequences with at most `CR_DECIMAL_DIG` significant decimal digits, the value of `CR_DECIMAL_DIG` shall be at least `M + 3`, where `M` is the maximum value of the `T_DECIMAL_DIG` macros for IEC 60559 binary formats. If the implementation correctly rounds for all numbers of significant decimal digits, then `CR_DECIMAL_DIG` shall have the value of the macro `UINTMAX_MAX`.

2. **Conversions involving** floating-point **types** are recommended practice. In particular, conversion between any supported binary formats to character sequences with more than `CR_DECIMAL_DIG` significant decimal digits shall correctly round to `CR_DECIMAL_DIG` significant digits and pad zeros on the right.

3. **Conversions from character sequences** with more than `CR_DECIMAL_DIG` significant decimal digits to **types** with IEC 60559 format and decimal with `DECIMAL_DIG` or fewer significant digits is correctly rounded (honor the current rounding mode), which assures that conversion from the widest supported binary formats shall correctly round to an intermediate character sequence with `CR_DECIMAL_DIG` significant decimal digits, according to the applicable rounding direction, and correctly round the intermediate result (having `CR_DECIMAL_DIG` significant decimal digits) to the destination type. The “inexact” floating-point exception is raised (once) if either conversion is inexact. (The second conversion may raise the “overflow” or “underflow” floating-point exception.)

---

384 IEC 60559 recommends that implicit floating-to-integer conversions raise the “inexact” floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the “inexact” floating-point exception. See `<math.h>`.

385 The intermediate conversion is exact only if all input digits after the first `CR_DECIMAL_DIG` digits are 0.

<table>
<thead>
<tr>
<th>IEC 60559 operation</th>
<th>C function</th>
<th>Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan</td>
<td>atan</td>
<td>7.12.4.3, F.10.1.3</td>
</tr>
<tr>
<td>atan2</td>
<td>atan2</td>
<td>7.12.4.4, F.10.1.4</td>
</tr>
<tr>
<td>sinh</td>
<td>sinh</td>
<td>7.12.5.5, F.10.2.5</td>
</tr>
<tr>
<td>cosh</td>
<td>cosh</td>
<td>7.12.5.4, F.10.2.4</td>
</tr>
<tr>
<td>tanh</td>
<td>tanh</td>
<td>7.12.5.6, F.10.2.6</td>
</tr>
<tr>
<td>asinh</td>
<td>asinh</td>
<td>7.12.5.2, F.10.2.2</td>
</tr>
<tr>
<td>acosh</td>
<td>acosh</td>
<td>7.12.5.1, F.10.2.1</td>
</tr>
<tr>
<td>atanh</td>
<td>atanh</td>
<td>7.12.5.3, F.10.2.3</td>
</tr>
</tbody>
</table>
The specification in this subclause assures conversion between IEC 60559 binary format and
decimal character sequence follows all pertinent recommended practice. It also assures
collection from IEC 60559 format to decimal with DECIMAL_DIG character sequence with at
least T_DECIMAL_DIG digits and back, using to-nearest rounding, is the identity function, where T
is the macro prefix for the format.

Functions such as `strtof` that convert character sequences to floating types honor the round-
ing direction. Hence, if the rounding direction might be upward or downward, the implementation
cannot convert a minus-signed sequence by negating the converted unsigned sequence.

NOTE IEC 60559 specifies that conversion to one-digit character strings using roundTiesToEven when both choices have
an odd least significant digit shall produce the value with the larger magnitude. For example, this can happen with 9.5e2
whose nearest neighbors are 9.62 and 1.63, both of which have a single odd digit in the significand part.

F.6 The return statement
If the return expression is evaluated in a floating-point format different from the return type, the
expression is converted as if by assignment to the return type of the function and the resulting
value is returned to the caller.

F.7 Contracted expressions
A contracted expression should raise floating-point exceptions in a manner generally consistent
with the basic arithmetic operations.

F.8 Floating-point environment
The floating-point environment defined in `<fenv.h>` includes the IEC 60559 floating-point exception
status flags and directed-rounding control modes. It includes also IEC 60559 dynamic rounding
precision and trap enablement modes, if the implementation supports them.

F.8.1 Environment management
IEC 60559 requires that floating-point operations implicitly raise floating-point exception status
flags, and that rounding control modes can be set explicitly to affect result values of floating-point
operations. When the state for the FENV_ACCESS pragma (defined in ) is “on”, these changes to the floating-point state are treated as side effects which respect which respect sequence points.

F.8.2 Translation
During translation, constant rounding direction modes (7.6.2) are in effect where specified. Elsewhere, during translation the IEC 60559 default modes are in effect:

- The rounding direction mode is rounding to nearest.
- The rounding precision mode (if supported) is set so that results are not shortened.
- Trapping or stopping (if supported) is disabled on all floating-point exceptions.

Recommended practice
The implementation should produce a diagnostic message for each translation-time floating-point
exception, other than “inexact”, the implementation should then proceed with the translation of

Assignment removes any extra range and precision.
This specification does not require dynamic rounding precision nor trap enablement modes.
If the state for the FENV_ACCESS pragma is “off”, the implementation is free to assume the dynamic floating-point control
modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see E9).
As floating constants are converted to appropriate internal representations at translation time, their conversion is subject
to constant or default rounding modes and raises no execution-time floating-point exceptions (even where the state of the

448 IEC 60559 floating-point arithmetic § F.8.2
the program.

F.8.3 Execution

At program startup the dynamic floating-point environment is initialized as prescribed by IEC 60559:

- All floating-point exception status flags are cleared.
- The dynamic rounding direction mode is rounding to nearest.
- The dynamic rounding precision mode (if supported) is set so that results are not shortened.
- Trapping or stopping (if supported) is disabled on all floating-point exceptions.

F.8.4 Constant expressions

An arithmetic constant expression of floating type, other than one in an initializer for an object that has static or thread storage duration, is evaluated (as if) during execution; thus, it is affected by any operative floating-point control modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the `FENV_ACCESS` pragma is “on”).

**EXAMPLE**

```c
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
    float w[3] = { 0.0/0.0 };  // raises an exception
    static float x = 0.0/0.0;  // does not raise an exception
    float y = 0.0/0.0;          // raises an exception
    double z = 0.0/0.0;         // raises an exception
    /* ... */
}
```

For the static initialization, the division is done at translation time, raising no (execution-time) floating-point exceptions. On the other hand, for the three automatic initializations the invalid division occurs at execution time.

F.8.5 Initialization

All computation for automatic initialization is done (as if) at execution time; thus, it is affected by any operative modes and raises floating-point exceptions as required by IEC 60559 (provided the state for the `FENV_ACCESS` pragma is “on”). All computation for initialization of objects that have static or thread storage duration is done (as if) at translation time.

**EXAMPLE**

```c
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(void)
{
    float u[3] = { 1.1e75 };  // raises exceptions
    static float v = 1.1e75;  // does not raise exceptions
    float w = 1.1e75;         // raises exceptions
    double x = 1.1e75;        // may raise exceptions
    float y = 1.1e75f;        // may raise exceptions
    long double z = 1.1e75;   // does not raise exceptions
    /* ... */
}
```

Where the state for the `FENV_ACCESS` pragma is “on”), Library functions, for example `strtod`, provide execution-time conversion of numeric strings. 

390] Where the state for the `FENV_ACCESS` pragma is “on”), results of inexact expressions like `1.0/3.0` are affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` generate execution-time floating-point exceptions.

The programmer can achieve the efficiency of translation-time evaluation through static initialization, such as

```c
const static double one_third = 1.0/3.0;
```
3 The static initialization of \( v \) raises no (execution-time) floating-point exceptions because its computation is done at translation time. The automatic initialization of \( u \) and \( w \) require an execution-time conversion to \texttt{float} of the wider value \( 1.1e75 \), which raises floating-point exceptions. The automatic initializations of \( x \) and \( y \) entail execution-time conversion; however, in some expression evaluation methods, the conversions is not to a narrower format, in which case no floating-point exception is raised\(^{391}\). The automatic initialization of \( z \) entails execution-time conversion, but not to a narrower format, so no floating-point exception is raised. Note that the conversions of the floating constants \( 1.1e75 \) and \( 1.1e75f \) to their internal representations occur at translation time in all cases.

### F.8.6 Changing the environment

1 Operations defined in \( \textsf{6.5} \) and functions and macros defined for the standard libraries change floating-point status flags and control modes just as indicated by their specifications (including conformance to IEC 60559). They do not change flags or modes (so as to be detectable by the user) in any other cases.

2 If the argument to the \texttt{feraiseexcept} function in \texttt{<fenv.h>} represents IEC 60559 valid coincident floating-point exceptions for atomic operations (namely “overflow” and “inexact”, or “underflow” and “inexact”), then “overflow” or “underflow” is raised before “inexact”.

### F.9 Optimization

1 This section identifies code transformations that might subvert IEC 60559-specified behavior, and others that do not.

#### F.9.1 Global transformations

1 Floating-point arithmetic operations and external function calls may entail side effects which optimization shall honor, at least where the state of the \texttt{FENV_ACCESS} pragma is “on”. The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations (\( +, * \), etc.) implicitly read the modes and write the flags.

2 Concern about side effects may inhibit code motion and removal of seemingly useless code. For example, in

```c
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
void f(double x)
{
   /* .... */
   for (i = 0; i < n; i++) x + 1;
   /* .... */
}
```

\( x+1 \) might raise floating-point exceptions, so cannot be removed. And since the loop body might not execute (maybe \( 0 \geq n \)), \( x+1 \) cannot be moved out of the loop. (Of course these optimizations are valid if the implementation can rule out the nettlesome cases.)

3 This specification does not require support for trap handlers that maintain information about the order or count of floating-point exceptions. Therefore, between function calls, floating-point exceptions need not be precise: the actual order and number of occurrences of floating-point exceptions (> 1) may vary from what the source code expresses. Thus, the preceding loop could be treated as

```c
if (0 < n) x + 1;
```

\(^{391}\)Use of \texttt{float_t} and \texttt{double_t} variables increases the likelihood of translation-time computation. For example, the automatic initialization

```c
double_t x = 1.1e75;
```

could be done at translation time, regardless of the expression evaluation method.
F9.2 Expression transformations

1 Valid expression transformations must preserve numerical values.

2 The equivalences noted below apply to expressions of standard floating types.

\[
x/2 \leftrightarrow x \times 0.5
\]

Although similar transformations involving inexact constants generally do not yield numerically equivalent expressions, if the constants are exact then such transformations can be made on IEC 60559 machines and others that round perfectly.

\[1 \times x \text{ and } x/1 \rightarrow x\]  The expressions \(1 \times x\), \(x/1\), and \(x\) are may be regarded as equivalent (on IEC 60559 machines, among others).\(^{392}\)

\[\text{DOLLAR}DIFx/x\]

\[
x/x \rightarrow 1.0
\]

The expressions \(x/x\) and 1.0 are not equivalent if \(x\) can be zero, infinite, or NaN.

\[\text{DOLLAR}DIFx/y\]

\[
x - y \leftrightarrow x + (-y)
\]

The expressions \(x - y\), \(x + (-y)\), and \((-y) + x\) are equivalent (on IEC 60559 machines, among others).

\[\text{DOLLAR}DIFy/y\]

\[
x - y \leftrightarrow -(y - x)
\]

The expressions \(x - y\) and \(-(y - x)\) are not equivalent because \(1 - 1\) is \(+0\) but \(-(1 - 1)\) is \(-0\) (in the default rounding direction).\(^{393}\)

\[
x - x \rightarrow 0.0
\]

The expressions \(x - x\) and 0.0 are not equivalent if \(x\) is a NaN or infinite.

\[
0 \times x \rightarrow 0.0
\]

The expressions \(0 \times x\) and 0.0 are not equivalent if \(x\) is a NaN, infinite, or \(-0\).

\[
x + 0 \rightarrow x
\]

The expressions \(x + 0\) and \(x\) are not equivalent if \(x\) is \(-0\), because \((-0) + (+0)\) yields \(+0\) (in the default rounding direction), not \(-0\).

\[
x - 0 \rightarrow x
\]

\((+0) - (+0)\) yields \(-0\) when rounding is downward (toward \(-\infty\)), but \(+0\) otherwise, and \((-0) - (+0)\) always yields \(-0\); so, if the state of the FENV_ACCESS pragma is “off”, promising default rounding, then the implementation can replace \(x - 0\) by \(x\), even if \(x\) might be zero.

\[
-x \leftrightarrow 0 - x
\]

The expressions \(-x\) and \(0 - x\) are not equivalent if \(x\) is \(+0\), because \((-+0)\) yields \(-0\), but \(0 - (+0)\) yields \(+0\) (unless rounding is downward).

3 For expressions of decimal floating types, transformations must preserve quantum exponents, as well as numerical values (5.2.4.2.5).

4 EXAMPLE 1 \times x \rightarrow x \text{ is valid for decimal floating-point expressions } x, \text{ but } 1.0 \times x \rightarrow x \text{ is not.}

\[
1.0 \times 2.34 = (1.0, 0) \times (1.0, 234, -2) = (1.0, 234, -2) = 12.34
\]

\[
1.0 \times 12.34 = (1.10, -1) \times (1.0, 234, -2) = (1.1234, -3) = 12.340
\]

The results are numerically equal, but have different quantum exponents, hence have different values.

\(^{392}\)Strict support for signaling NaNs — not required by this specification — would invalidate these and other transformations that remove arithmetic operators.

\(^{393}\)IEC 60559 prescribes a signed zero to preserve mathematical identities across certain discontinuities. Examples include:

\[
1/(1/\pm \infty) \text{ is } \pm \infty
\]

\[
\text{and } \text{conj(csrz}(x)) \text{ is csqrt}(\text{conj}(x)),
\]

for complex \(x\).
F.9.3 Relational operators

1. \( x \neq x \rightarrow \texttt{false} \)  
   The expression \( x \neq x \) is true if \( x \) is a NaN.

2. \( x = x \rightarrow \texttt{true} \)  
   The expression \( x = x \) is false if \( x \) is a NaN.

3. \( x < y \rightarrow \texttt{isless}(x,y) \) (and similarly for \( \leq, >, \geq \)) Though numerically equal, these expressions are not equivalent because of side effects when \( x \) or \( y \) is a NaN and the state of the \texttt{FENV_ACCESS} pragma is “on”. This transformation, which would be desirable if extra code were required to cause the “invalid” floating-point exception for unordered cases, could be performed provided the state of the \texttt{FENV_ACCESS} pragma is “off”.

The sense of relational operators shall be maintained. This includes handling unordered cases as expressed by the source code.

2 EXAMPLE

```c
// calls g and raises “invalid” if a and b are unordered
if (a < b)
    f();
else
    g();
```

is not equivalent to

```c
// calls f and raises “invalid” if a and b are unordered
if (a >= b)
    g();
else
    f();
```

nor to

```c
// calls f without raising “invalid” if a and b are unordered
if (isgreaterequal(a,b))
    g();
else
    f();
```

nor, unless the state of the \texttt{FENV_ACCESS} pragma is “off”, to

```c
// calls g without raising “invalid” if a and b are unordered
if (isless(a,b))
    f();
else
    g();
```

but is equivalent to

```c
if (! (a < b))
    g();
else
    f();
```

F.9.4 Constant arithmetic

1 The implementation shall honor floating-point exceptions raised by execution-time constant arithmetic wherever the state of the \texttt{FENV_ACCESS} pragma is “on”. (See F.8.4 and F.8.5.) An operation on constants that raises no floating-point exception can be folded during translation, except, if the state of the \texttt{FENV_ACCESS} pragma is “on”, a further check is required to assure that
changing the rounding direction to downward does not alter the sign of the result,\(^\text{394}\) and implementations that support dynamic rounding precision modes shall assure further that the result of the operation raises no floating-point exception when converted to the semantic type of the operation.

### F.10 Mathematics `<math.h>` and `<tgmath.h>`

1. This subclause contains specifications of `<math.h>` and `<tgmath.h>` facilities that are particularly suited for IEC 60559 implementations.

2. The Standard C macro `HUGE_VAL` and its `float` and `long double` analogs, `HUGE_VALF` and `HUGE_VALL`, expand to expressions whose values are positive infinities.

3. For each single-argument function `f` in `<math.h>` whose mathematical counterpart is symmetric (even), `f(-x)` is `f(x)` for all rounding modes and for all `x` in the (valid) domain of the function. For each single-argument function `f` in `<math.h>` whose mathematical counterpart is antisymmetric (odd), `f(-x)` is `-f(x)` for the IEC 60559 rounding modes `roundTiesToEven`, `roundTiesToAway`, and `roundTowardZero`, and for all `x` in the (valid) domain of the function. The `atan2` and `atan2pi` functions are odd in their first argument.

4. Special cases for functions in `<math.h>` are covered directly or indirectly by IEC 60559. The functions that IEC 60559 specifies directly are identified in 22F.3. The other functions in `<math.h>` treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the `FENV_ACCESS` pragma is “on”) the floating-point status flags in a manner consistent with the basic arithmetic operations covered by IEC 60559—IEC 60559 operations.

5. The expression `math_errhandling` & `MATH_ERREXCEPT` shall evaluate to a nonzero value.

6. The functions bound to operations in IEC 60559 (F.3) are fully specified by IEC 60559, including rounding behaviors and floating-point exceptions.

7. The “invalid” and “divide-by-zero” floating-point exceptions are raised as specified in subsequent subclauses of this annex.

8. The “overflow” floating-point exception is raised whenever an infinity — or, because of rounding direction, a maximal-magnitude finite number — is returned in lieu of a value whose magnitude is too large.

9. The “underflow” floating-point exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy.\(^\text{395}\)

10. Whether or when library functions not bound to operations in IEC 60559 raise the “inexact” floating-point exception is unspecified, unless explicitly specified stated otherwise.

11. Whether or when library functions raise an undeserved “underflow” floating-point exception is unspecified.\(^\text{396}\) Otherwise, as implied by F.8.6, the F.8.6, these functions do not raise spurious floating-point exceptions (detectable by the user), other than the “inexact” floating-point exception.

12. Whether the functions not bound to operations in IEC 60559 honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

13. Functions with a NaN argument return a NaN result and raise no floating-point exception, except where explicitly stated otherwise.

14. The specifications in the following subclauses append to the definitions in `<math.h>`. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol “±” occurs in both an argument and the result, the result has the same sign as the argument.

---

\(^{394}\) 0 \(-0\) yields \(-0\) instead of \(+0\) just when the rounding direction is downward.

\(^{395}\) IEC 60559 allows different definitions of underflow. They all result in the same values, but differ on when the floating-point exception is raised.

\(^{396}\) It is intended that undeserved “underflow” and “inexact” floating-point exceptions are raised only if avoiding them would be too costly.
Recommended practice

If a function with one or more NaN arguments returns a NaN result, the result should be the same as one of the NaN arguments (after possible type conversion), except perhaps for the sign.

F.10.1 Trigonometric functions

F.10.1.1 The acos functions

1. \( \text{acos}(1) \) returns +0.
2. \( \text{acos}(x) \) returns a NaN and raises the “invalid” floating-point exception for \( |x| > 1 \).

F.10.1.2 The asin functions

1. \( \text{asin}(\pm 0) \) returns \( \pm 0 \).
2. \( \text{asin}(x) \) returns a NaN and raises the “invalid” floating-point exception for \( |x| > 1 \).

F.10.1.3 The atan functions

1. \( \text{atan}(\pm 0) \) returns \( \pm 0 \).
2. \( \text{atan}(\pm \infty) \) returns \( \pm \pi/2 \) for \( x < 0 \).
3. \( \text{atan}(\pm 0, x) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for \( x > 0 \).
4. \( \text{atan}(y, \pm 0) \) returns \( \pm \pi/2 - \frac{\pi}{4} \) for \( y < 0 \).
5. \( \text{atan}(y, \pm 0) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for \( y > 0 \).
6. \( \text{atan}(\pm y, -\infty) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for finite \( x > 0 \).
7. \( \text{atan}(\pm y, +\infty) \) returns \( \pm \pi/2 - \frac{\pi}{4} \) for finite \( x > 0 \).
8. \( \text{atan}(\pm \infty, x) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for finite \( x > 0 \).
9. \( \text{atan}(\pm \infty, -\infty) \) returns \( \pm \pi/2 - \frac{\pi}{4} \).

F.10.1.4 The atan2 functions

1. \( \text{atan2}(\pm 0, -0) \) returns \( \pm \pi \).
2. \( \text{atan2}(\pm 0, +0) \) returns \( \pm 0 \).
3. \( \text{atan2}(\pm 0, x) \) returns \( \pm \pi \) for \( x < 0 \).
4. \( \text{atan2}(\pm 0, x) \) returns \( \pm 0 \) for \( x > 0 \).
5. \( \text{atan2}(y, \pm 0) \) returns \( \pm \pi/2 - \frac{\pi}{4} \) for \( y < 0 \).
6. \( \text{atan2}(y, \pm 0) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for \( y > 0 \).
7. \( \text{atan2}(\pm y, -\infty) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for finite \( y < 0 \).
8. \( \text{atan2}(\pm y, +\infty) \) returns \( \pm \pi/2 - \frac{\pi}{4} \) for finite \( y > 0 \).
9. \( \text{atan2}(\pm \infty, x) \) returns \( \pm \pi/2 + \frac{\pi}{4} \) for finite \( x > 0 \).
10. \( \text{atan2}(\pm \infty, -\infty) \) returns \( \pm \pi/2 - \frac{\pi}{4} \).

F.10.1.5 The cos functions

1. \( \text{cos}(\pm 0) \) returns 1.
2. \( \text{cos}(\pm \infty) \) returns a NaN and raises the “invalid” floating-point exception.

F.10.1.6 The sin functions

1. \( \text{sin}(\pm 0) \) returns 0.
2. \( \text{sin}(\pm \infty) \) returns a NaN and raises the “invalid” floating-point exception.

F.10.1.7 The tan functions

1. \( \text{tan}(\pm 0) \) returns 0.
2. \( \text{tan}(\pm \infty) \) returns a NaN and raises the “invalid” floating-point exception.

\[^{397}\text{atan2}(0, 0) \text{ does not raise the “invalid” floating-point exception, nor does } \text{atan2}(y, 0) \text{ raise the “divide-by-zero” floating-point exception.}\]
F.10.1.8 The \texttt{acospi} functions

1. \texttt{acospi}(+1) returns +0.

2. \texttt{acospi}(x) returns a NaN and raises the “invalid” floating-point exception for \(|x| > 1|.

F.10.1.9 The \texttt{asinpi} functions

1. \texttt{asinpi}(0) returns ±0.

2. \texttt{asinpi}(x) returns a NaN and raises the “invalid” floating-point exception for \(|x| > 1|.

F.10.1.10 The \texttt{atanpi} functions

1. \texttt{atanpi}(0) returns ±0.

2. \texttt{atanpi}(±\infty) returns ±\frac{\pi}{2}.

F.10.1.11 The \texttt{atan2pi} functions

1. \texttt{atan2pi}(±0, 0) returns +1.\footnote{\texttt{atan2pi}(0, 0) does not raise the “invalid” floating-point exception, nor does \texttt{atan2pi}(y, 0) raise the “divide-by-zero” floating-point exception.}

- \texttt{atan2pi}(±0, +0) returns ±0.
- \texttt{atan2pi}(±0, x) returns ±1 for \(x < 0|.
- \texttt{atan2pi}(±0, x) returns ±0 for \(x > 0|.
- \texttt{atan2pi}(y, ±0) returns ±\frac{\pi}{2} for \(y < 0|.
- \texttt{atan2pi}(y, ±0) returns ±\frac{\pi}{2} for \(y > 0|.
- \texttt{atan2pi}(±y, ±\infty) returns ±1 for finite \(y > 0|.
- \texttt{atan2pi}(±y, +\infty) returns ±0 for finite \(y > 0|.
- \texttt{atan2pi}(±\infty, x) returns ±\frac{\pi}{2} for finite \(x|.
- \texttt{atan2pi}(±\infty, ±\infty) returns ±\frac{\pi}{2}.
- \texttt{atan2pi}(±\infty, +\infty) returns ±\frac{\pi}{2}.

F.10.1.12 The \texttt{cospi} functions

1. \texttt{cospi}(0) returns 1.

- \texttt{cospi}(n + \frac{1}{4}) returns +0, for integers \(n|.
- \texttt{cospi}(±\infty) returns a NaN and raises the “invalid” floating-point exception.

F.10.1.13 The \texttt{sinpi} functions

1. \texttt{sinpi}(0) returns ±0.

- \texttt{sinpi}(n) returns ±0, for positive integers \(n|.
- \texttt{sinpi}(±\infty) returns a NaN and raises the “invalid” floating-point exception.
F.10.14 The \texttt{tanpi} functions

\begin{itemize}
\item \texttt{tanpi}(\pm 0) returns \pm 0.
\item \texttt{tanpi}(n) returns \pm 0, for positive even and negative odd integers \( n \).
\item \texttt{tanpi}(n) returns \pm 0, for positive odd and negative even integers \( n \).
\item \texttt{tanpi}(n + \frac{1}{2}) returns \pm \infty and raises the “divide-by-zero” floating-point exception, for even integers \( n \).
\item \texttt{tanpi}(n + \frac{1}{2}) returns \pm \infty and raises the “divide-by-zero” floating-point exception, for odd integers \( n \).
\item \texttt{tanpi}(\pm \infty) returns a NaN and raises the “invalid” floating-point exception.
\end{itemize}

F.10.2 Hyperbolic functions

F.10.2.1 The \texttt{acosh} functions

\begin{itemize}
\item \texttt{acosh}(1) returns +0.
\item \texttt{acosh}(x) returns a NaN and raises the “invalid” floating-point exception for \( x < 1 \).
\item \texttt{acosh}(\pm \infty) returns +\infty.
\end{itemize}

F.10.2.2 The \texttt{asinh} functions

\begin{itemize}
\item \texttt{asinh}(\pm 0) returns \pm 0.
\item \texttt{asinh}(\pm \infty) returns \pm \infty.
\end{itemize}

F.10.2.3 The \texttt{atanh} functions

\begin{itemize}
\item \texttt{atanh}(\pm 0) returns \pm 0.
\item \texttt{atanh}(\pm 1) returns \pm \infty and raises the “divide-by-zero” floating-point exception.
\item \texttt{atanh}(x) returns a NaN and raises the “invalid” floating-point exception for \( |x| > 1 \).
\end{itemize}

F.10.2.4 The \texttt{cosh} functions

\begin{itemize}
\item \texttt{cosh}(\pm 0) returns 1.
\item \texttt{cosh}(\pm \infty) returns +\infty.
\end{itemize}

F.10.2.5 The \texttt{sinh} functions

\begin{itemize}
\item \texttt{sinh}(\pm 0) returns \pm 0.
\item \texttt{sinh}(\pm \infty) returns \pm \infty.
\end{itemize}

F.10.2.6 The \texttt{tanh} functions

\begin{itemize}
\item \texttt{tanh}(\pm 0) returns \pm 0.
\item \texttt{tanh}(\pm \infty) returns \pm 1.
\end{itemize}

F.10.3 Exponential and logarithmic functions

F.10.3.1 The \texttt{exp} functions

\begin{itemize}
\item \texttt{exp}(\pm 0) returns 1.
\item \texttt{exp}(\pm \infty) returns +\infty.
\end{itemize}
F.10.3.2 The \texttt{exp10} functions

1. \texttt{exp10}(\pm 0) returns 1.
2. \texttt{exp10}(\infty) returns +0.
3. \texttt{exp10}(-\infty) returns +\infty.

F.10.3.3 The \texttt{exp10m1} functions

1. \texttt{exp10m1}(\pm 0) returns \pm 0.
2. \texttt{exp10m1}(\infty) returns -1.
3. \texttt{exp10m1}(-\infty) returns +\infty.

F.10.3.4 The \texttt{exp2} functions

1. \texttt{exp2}(\pm 0) returns 1.
2. \texttt{exp2}(\infty) returns +0.
3. \texttt{exp2}(-\infty) returns +\infty.

F.10.3.5 The \texttt{exp2m1} functions

1. \texttt{exp2m1}(\pm 0) returns \pm 0.
2. \texttt{exp2m1}(\infty) returns -1.
3. \texttt{exp2m1}(-\infty) returns +\infty.

F.10.3.6 The \texttt{expm1} functions

1. \texttt{expm1}(\pm 0) returns \pm 0.
2. \texttt{expm1}(\infty) returns -1.
3. \texttt{expm1}(-\infty) returns +\infty.

F.10.3.7 The \texttt{frexp} functions

1. \texttt{frexp}(\pm 0, \texttt{exp}) returns \pm 0, and stores 0 in the object pointed to by \texttt{exp}.
2. \texttt{frexp}(\pm\infty, \texttt{exp}) returns \pm\infty, and stores an unspecified value in the object pointed to by \texttt{exp}.
3. \texttt{frexp}(\texttt{NaN}, \texttt{exp}) stores an unspecified value in the object pointed to by \texttt{exp} (and returns a \texttt{NaN}).
4. \texttt{frexp} raises no floating-point exceptions if \texttt{value} is not a signaling NaN.
5. The returned value is independent of the current rounding direction mode.
6. On a binary system, the body of the \texttt{frexp} function might be

\[
\begin{cases}
\ast \texttt{exp} = (\texttt{value} == 0) \ ? \ 0 : (\texttt{int})(1 + \logb(\texttt{value})); \\
\text{return scalbn(\texttt{value}, -(\ast \texttt{exp}))};
\end{cases}
\]

F.10.3.8 The \texttt{ilogb} functions

1. When the correct result is representable in the range of the return type, the returned value is exact and is independent of the current rounding direction mode.
2. If the correct result is outside the range of the return type, the numeric result is unspecified and the “invalid” floating-point exception is raised.
3. \texttt{ilogb}(x), for \texttt{x} zero, infinite, or \texttt{NaN}, raises the “invalid” floating-point exception and returns the value specified in 7.12.6.8.
F.10.3.9 The \texttt{ldexp} functions

On a binary system, \texttt{ldexp}(x, \, \texttt{exp}) is equivalent to \texttt{scalbn}(x, \, \texttt{exp}).

F.10.3.10 The \texttt{llogb} functions

The \texttt{llogb} functions are equivalent to the \texttt{ilogb} functions, except that the \texttt{llogb} functions determine a result in the \texttt{long int} type.

F.10.3.11 The \texttt{log} functions

- \texttt{\log}(±0) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log}(1) returns 0.
- \texttt{\log}(x) returns a NaN and raises the “invalid” floating-point exception for $x < 0$.
- \texttt{\log}(+\infty) returns $+\infty$.

F.10.3.12 The \texttt{log10} functions

- \texttt{\log10}(±0) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log10}(1) returns 0.
- \texttt{\log10}(x) returns a NaN and raises the “invalid” floating-point exception for $x < 0$.
- \texttt{\log10}(+\infty) returns $+\infty$.

F.10.3.13 The \texttt{log10p1} functions

- \texttt{\log1p}(±0)\texttt{\log10p1}(±0) returns ±0.
- \texttt{\log10p1}(−1) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log10p1}(x) returns a NaN and raises the “invalid” floating-point exception for $x < -1$.
- \texttt{\log10p1}(+\infty) returns $+\infty$.

F.10.3.14 The \texttt{log1p} and \texttt{log1} functions

- \texttt{\log1p}(±0) returns ±0.
- \texttt{\log1p}(−1)\texttt{\log1p1}(−1) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log1p}(x) returns a NaN and raises the “invalid” floating-point exception for $x < -1$.
- \texttt{\log1p}(+\infty)\texttt{\log1p1}(+\infty) returns $+\infty$.

The \texttt{log1p} functions are equivalent to the \texttt{log1} functions.

F.10.3.15 The \texttt{log2} functions

- \texttt{\log2}(±0) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log2}(1) returns 0.
- \texttt{\log2}(x) returns a NaN and raises the “invalid” floating-point exception for $x < 0$.
- \texttt{\log2}(+\infty) returns $+\infty$.

F.10.3.16 The \texttt{log2p1} functions

- \texttt{\log2p1}(±0) returns ±0.
- \texttt{\log2p1}(−1) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
- \texttt{\log2p1}(x) returns a NaN and raises the “invalid” floating-point exception for $x < -1$.
- \texttt{\log2p1}(+\infty) returns $+\infty$.
F.10.3.17 The logb functions
1 — \texttt{logb}(0) returns $-\infty$ and raises the “divide-by-zero” floating-point exception.
   — \texttt{logb}(\pm\infty) returns $+\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.3.18 The modf functions
1 — \texttt{modf}(\pm x, \texttt{iptr}) returns a result with the same sign as $x$.
   — \texttt{modf}(\pm\infty, \texttt{iptr}) returns $\pm 0$ and stores $\pm\infty$ in the object pointed to by \texttt{iptr}.
   — \texttt{modf}(\texttt{NaN}, \texttt{iptr}) stores a NaN in the object pointed to by \texttt{iptr}.

2 The returned values are exact and are independent of the current rounding direction mode.

3 \texttt{modf} behaves as though implemented by

\begin{verbatim}
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

double modf(double value, double *iptr)
{
    int save_round = fegetround();
    fesetround(FE_TOWARDZERO);
    *iptr = nearbyint(value);
    fesetround(save_round);
    return copysign(isinf(value) ? 0.0:
                        value - (*iptr), value);
}
\end{verbatim}

F.10.3.19 The scalbn and scalbln functions
1 — \texttt{scalbn}(0, \texttt{n}) returns $0$.
   — \texttt{scalbn}(\texttt{x}, 0) returns $x$.
   — \texttt{scalbn}(\pm\infty, \texttt{n}) returns $\pm\infty$.

2 If the calculation does not overflow or underflow, the returned value is exact and independent of the current rounding direction mode.

F.10.4 Power and absolute value functions

F.10.4.1 The cbrt functions
1 — \texttt{cbrt}(0) returns $0$.
   — \texttt{cbrt}(\pm\infty) returns $\pm\infty$.

F.10.4.2 The compoundn functions
1 — \texttt{compound}(\pm 0, \texttt{n}) returns $1$ for $x \geq -1$ or $x$ a NaN.
   — \texttt{compound}(\texttt{x}, 0) returns $x$.
   — \texttt{compound}(\pm\infty, \texttt{n}) returns $\pm\infty$ and raises the “invalid” floating-point exception for $x \leq -1$.
   — \texttt{compound}(\texttt{nan}, \texttt{n}) returns a NaN and raises the “invalid” floating-point exception for $x \leq -1$.
   — \texttt{compound}(\pm\infty, \texttt{n}) returns $\pm\infty$ and raises the divide-by-zero floating-point exception for $n \leq 0$.
   — \texttt{compound}(\pm 0, \texttt{n}) returns $1$ for $n > 0$.

§ F.10.4.2 IEC 60559 floating-point arithmetic 459
F.10.4.3 The \texttt{fabs} functions

1. \texttt{fabs}(±0) returns +0.  
2. \texttt{fabs}(±∞) returns +∞.

\texttt{fabs}(x) raises no floating-point exceptions, even if \(x\) is a signaling NaN. The returned value is exact and is independent of the current rounding direction mode.

F.10.4.4 The \texttt{hypot} functions

1. \texttt{hypot}(x, y), \texttt{hypot}(y, x), and \texttt{hypot}(x, −y) are equivalent.
2. \texttt{hypot}(x, ±0) is equivalent to \texttt{fabs}(x).
3. \texttt{hypot}(±∞, y) returns +∞, even if \(y\) is a NaN.

F.10.4.5 The \texttt{pow} functions

1. \texttt{pow}(±0, y) returns ±∞ and raises the “divide-by-zero” floating-point exception for \(y\) an odd integer < 0.
2. \texttt{pow}(±0, y) returns +∞ and raises the “divide-by-zero” floating-point exception for \(y < 0\), finite, and not an odd integer.
3. \texttt{pow}(±0, −∞) returns +∞ and may raise the “divide-by-zero” floating-point exception.
4. \texttt{pow}(±0, y) returns ±0 for \(y\) an odd integer > 0.
5. \texttt{pow}(±0, y) returns +0 for \(y > 0\) and not an odd integer.
6. \texttt{pow}(−1, ±∞) returns 1.
7. \texttt{pow}(+1, y) returns 1 for any \(y\), even a NaN.
8. \texttt{pow}(x, ±0) returns 1 for any \(x\), even a NaN.
9. \texttt{pow}(x, y) returns a NaN and raises the “invalid” floating-point exception for finite \(x < 0\) and finite non-integer \(y\).
10. \texttt{pow}(x, −∞) returns +∞ for \(|x| < 1\).
11. \texttt{pow}(x, −∞) returns +0 for \(|x| > 1\).
12. \texttt{pow}(x, +∞) returns +0 for \(|x| < 1\).
13. \texttt{pow}(x, +∞) returns +∞ for \(|x| > 1\).
14. \texttt{pow}(−∞, y) returns −0 for \(y\) an odd integer < 0.
15. \texttt{pow}(−∞, y) returns +0 for \(y < 0\) and not an odd integer.
16. \texttt{pow}(−∞, y) returns −∞ for \(y\) an odd integer > 0.
17. \texttt{pow}(−∞, y) returns +∞ for \(y > 0\) and not an odd integer.
18. \texttt{pow}(+∞, y) returns +0 for \(y < 0\).
19. \texttt{pow}(+∞, y) returns +∞ for \(y > 0\).
F.10.4.6 The `pown` functions

1. `pown(x, 0)` returns 1 for all `x` not a signalling NaN.

2. `pown(±0, n)` returns `±∞` and raises the “divide-by-zero” floating-point exception for odd `n < 0`.

3. `pown(±0, n)` returns `+∞` and raises the “divide-by-zero” floating-point exception for even `n < 0`.

4. `pown(±0, n)` returns `+0` for even `n > 0`.

5. `pown(±0, n)` returns `-0` for odd `n > 0`.

6. `pown(±∞, n)` is equivalent to `pown(±0, -n)` for `n` not 0, except that the “divide-by-zero” floating-point exception is not raised.

F.10.4.7 The `powr` functions

1. `powr(x, ±0)` returns 1 for finite `x > 0`.

2. `powr(±0, y)` returns `+∞` and raises the “divide-by-zero” floating-point exception for finite `y < 0`.

3. `powr(±0, -∞)` returns `+∞`.

4. `powr(±0, y)` returns `+0` for `y > 0`.

5. `powr(±1, y)` returns 1 for finite `y`.

6. `powr(±1, y)` returns a NaN and raises the “invalid” floating-point exception for `x < 0`.

7. `powr(±0, ±0)` returns a NaN and raises the “invalid” floating-point exception.

8. `powr(±∞, ±0)` returns a NaN and raises the “invalid” floating-point exception.

F.10.4.8 The `rootn` functions

1. `rootn(±0, n)` returns `±∞` and raises the “divide-by-zero” floating-point exception for odd `n < 0`.

2. `rootn(±0, n)` returns `+∞` and raises the “divide-by-zero” floating-point exception for even `n < 0`.

3. `rootn(±0, n)` returns `+0` for even `n > 0`.

4. `rootn(±0, n)` returns `-0` for odd `n > 0`.

5. `rootn(±∞, n)` returns `+∞` for `n > 0`.

6. `rootn(−∞, n)` returns `-∞` for odd `n > 0`.

7. `rootn(−∞, n)` returns a NaN and raises the “invalid” floating-point exception for even `n > 0`.

8. `rootn(+∞, n)` returns `+0` for `n < 0`.

9. `rootn(−∞, n)` returns `-0` for odd `n < 0`.

10. `rootn(−∞, n)` returns a NaN and raises the “invalid” floating-point exception for even `n < 0`.

11. `rootn(x, 0)` returns a NaN and raises the “invalid” floating-point exception for all `x` (including NaN).

12. `rootn(x, n)` returns a NaN and raises the “invalid” floating-point exception for `x < 0` and `n` even.
F.10.4.9 The \texttt{rsqrt} functions

\begin{itemize}
  \item \texttt{rsqrt}(\pm 0) \text{ returns } \pm \infty \text{ and raises the “divide-by-zero” floating-point exception.}
  \item \texttt{rsqrt}(x) \text{ returns a NaN and raises the “invalid” floating-point exception for } x < 0.
  \item \texttt{rsqrt}(+\infty) \text{ returns } +0.
\end{itemize}

F.10.4.10 The \texttt{sqrt} functions

\begin{itemize}
  \item \texttt{sqrt}(\pm 0) \text{ returns } \pm 0.
  \item \texttt{sqrt}(+\infty) \text{ returns } +\infty.
  \item \texttt{sqrt}(x) \text{ returns a NaN and raises the “invalid” floating-point exception for } x < 0.
\end{itemize}

\texttt{sqrt} is fully specified as a basic arithmetic operation in IEC 60559. The returned value is dependent on the current rounding direction mode.

F.10.5 Error and gamma functions

F.10.5.1 The \texttt{erf} functions

\begin{itemize}
  \item \texttt{erf}(\pm 0) \text{ returns } \pm 0.
  \item \texttt{erf}(\pm \infty) \text{ returns } \pm 1.
\end{itemize}

F.10.5.2 The \texttt{erfc} functions

\begin{itemize}
  \item \texttt{erfc}(\infty) \text{ returns } 2.
  \item \texttt{erfc}(+\infty) \text{ returns } +0.
\end{itemize}

F.10.5.3 The \texttt{lgamma} functions

\begin{itemize}
  \item \texttt{lgamma}(1) \text{ returns } +0.
  \item \texttt{lgamma}(2) \text{ returns } +0.
  \item \texttt{lgamma}(x) \text{ returns } +\infty \text{ and raises the “divide-by-zero” floating-point exception for } x \text{ a negative integer or zero.}
  \item \texttt{lgamma}(\infty) \text{ returns } +\infty.
  \item \texttt{lgamma}(+\infty) \text{ returns } +\infty.
\end{itemize}

F.10.5.4 The \texttt{tgamma} functions

\begin{itemize}
  \item \texttt{tgamma}(\pm 0) \text{ returns } \pm \infty \text{ and raises the “divide-by-zero” floating-point exception.}
  \item \texttt{tgamma}(x) \text{ returns a NaN and raises the “invalid” floating-point exception for } x \text{ a negative integer.}
  \item \texttt{tgamma}(\infty) \text{ returns a NaN and raises the “invalid” floating-point exception.}
  \item \texttt{tgamma}(+\infty) \text{ returns } +\infty.
\end{itemize}

F.10.6 Nearest integer functions

F.10.6.1 The \texttt{ceil} functions

\begin{itemize}
  \item \texttt{ceil}(\pm 0) \text{ returns } \pm 0.
  \item \texttt{ceil}(\pm \infty) \text{ returns } \pm \infty.
\end{itemize}
The returned value is **exact** and is independent of the current rounding direction mode.

The **double** version of **ceil** behaves as though implemented by

```c
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

double ceil(double x)
{
    double result;
    int save_round = fegetround();
    fesetround(FE_UPWARD);
    result = nearbyint(x);
    fesetround(save_round);
    return result;
}
```

### F.10.6.2 The **floor** functions

1. **floor**(±0) returns ±0.
2. **floor**(±∞) returns ±∞.

3. The returned value is **exact** and is independent of the current rounding direction mode.

4. See the sample implementation for **ceil** in F.10.6.1. The **may**, but are not required to, raise the “inexact” floating-point exception for finite non-integer arguments, as this implementation does.

### F.10.6.3 The **nearbyint** functions

1. The **nearbyint** functions use IEC 60559 rounding according to the current rounding direction. They do not raise the “inexact” floating-point exception if the result differs in value from the argument.

   - **nearbyint**(±0) returns ±0 (for all rounding directions).
   - **nearbyint**(±∞) returns ±∞ (for all rounding directions).

### F.10.6.4 The **rint** functions

1. The **rint** functions differ from the **nearbyint** functions only in that they do raise the “inexact” floating-point exception if the result differs in value from the argument.

### F.10.6.5 The **lrint** and **llrint** functions

1. The **lrint** and **llrint** functions provide floating-to-integer conversion as prescribed by IEC 60559. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the “invalid” floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the “inexact” floating-point exception.

### F.10.6.6 The **round** functions

1. **round**(±0) returns ±0.
2. **round**(±∞) returns ±∞.

3. The returned value is independent of the current rounding direction mode.

4. The **double** version of **round** behaves as though implemented by

   ```c
   // This code does not handle signaling NaNs as required of implementations that define FE_SNANS_ALWAYS_SIGNAL
   ```

---

§ F.10.6.6 IEC 60559 floating-point arithmetic 463
```c
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

double round(double x)
{
    double result;
    fenv_t save_env;
    feholdexcept(&save_env);
    result = rint(x);
    if (fetestexcept(FE_INEXACT)) {
        fesetround(FE_TOWARDZERO);
        result = rint(copysign(0.5 + fabs(x), x));
        feclearexcept(FE_INEXACT);
    }
    feupdateenv(&save_env);
    return result;
}
```

F.10.6.7 The `lround` and `llround` functions

1. The `lround` and `llround` functions differ from the `rint` and `lrint` functions with the default rounding direction just in that the `lround` and `llround` functions round halfway cases away from zero and need not raise the “inexact” floating-point exception for non-integer arguments that round to within the range of the return type.

F.10.6.8 The `roundeven` functions

1. 
   - `roundeven(±0)` returns ±0.
   - `roundeven(±∞)` returns ±∞.

2. The returned value is exact and is independent of the current rounding direction mode.

3. See the sample implementation for `ceil` in F.10.6.1.

F.10.6.9 The `trunc` functions

1. The `trunc` functions use IEC 60559 rounding toward zero (regardless of the current rounding direction). The returned value is exact.

2. The returned value is exact and is independent of the current rounding direction mode. The may, but are not required to.

F.10.6.10 The `fromfp` and `ufromfp` functions

1. The `fromfp` and `ufromfp` functions raise the “invalid” floating-point exception and return an unspecified value if the floating-point argument `x` is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width.

2. These functions do not raise the “inexact” floating-point exception.

F.10.6.11 The `fromfpx` and `ufromfpx` functions

1. The `fromfpx` and `ufromfpx` functions raise the “invalid” floating-point exception and return an unspecified value if the floating-point argument `x` is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width.

2. These functions raise the “inexact” floating-point exception for finite non-integer arguments if a valid result differs in value from the floating-point argument `x`. 

464 IEC 60559 floating-point arithmetic § F.10.6.11
F.10.7 Remainder functions

F.10.7.1 The fmod functions

1. \( fmod(\pm 0, y) \) returns \( \pm 0 \) for \( y \) not zero.
2. \( fmod(x, y) \) returns a NaN and raises the “invalid” floating-point exception for \( x \) infinite or \( y \) zero (and neither is a NaN).
3. \( fmod(x, \pm \infty) \) returns \( x \) for \( x \) not infinite.

When subnormal results are supported, the returned value is exact and independent of the current rounding direction mode.

The double version of fmod behaves as though implemented by

```c
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON

double fmod(double x, double y) {
    double result;
    result = remainder(fabs(x), (y = fabs(y)));
    if (signbit(result)) result += y;
    return copysign(result, x);
}
```

F.10.7.2 The remainder functions

The remainder functions are fully specified as a basic arithmetic operation in IEC 60559.

1. \( \text{remainder}(\pm 0, y) \) returns \( \pm 0 \) for \( y \) not zero.
2. \( \text{remainder}(x, y) \) returns a NaN and raises the “invalid” floating-point exception for \( x \) infinite or \( y \) zero (and neither is a NaN).
3. \( \text{remainder}(x, \pm \infty) \) returns \( x \) for finite \( x \).

When subnormal results are supported, the returned value is exact and independent of the current rounding direction mode.

F.10.7.3 The remquo functions

The remquo functions follow the specifications for the remainder functions. They have no further specifications special to IEC 60559 implementations.

When subnormal results are supported, the returned value is exact and independent of the current rounding direction mode.

F.10.8 Manipulation functions

F.10.8.1 The copysign functions

copysign is specified in the Appendix to IEC 60559.

1. copysign\((x, y)\) raises no floating-point exceptions, even if \( x \) or \( y \) is a signaling NaN. The returned value is exact and independent of the current rounding direction mode.

F.10.8.2 The nan functions

1. All IEC 60559 implementations support quiet NaNs, in all floating formats.
2. The returned value is exact and independent of the current rounding direction mode.

F.10.8.3 The nextafter functions

1. nextafter\((x, y)\) raises the “overflow” and “inexact” floating-point exceptions for \( x \) finite and the function value infinite.
— `nextafter(x, y)` raises the “underflow” and “inexact” floating-point exceptions for the function value subnormal or zero and \( x \neq y \).

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

**F.10.8.4 The nexttoward functions**

1 No additional requirements beyond those on `nextafter`.

2 Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

**F.10.8.5 The nextup functions**

1 — `nextup(+\infty)` returns `+\infty`.

1 — `nextup(-\infty)` returns the largest-magnitude negative finite number in the type of the function.

2 `nextup(x)` raises no floating-point exceptions if \( x \) is not a signaling NaN. The returned value is independent of the current rounding direction mode.

**F.10.8.6 The nextdown functions**

1 — `nextdown(-\infty)` returns `−\infty`.

1 — `nextdown(+\infty)` returns the largest-magnitude positive finite number in the type of the function.

2 `nextdown(x)` raises no floating-point exceptions if \( x \) is not a signaling NaN. The returned value is independent of the current rounding direction mode.

**F.10.8.7 The canonicalize functions**

1 The canonicalize functions produce\(^{400}\) the canonical version of the representation in the object pointed to by the argument \( x \). If the input *\( x \) is a signaling NaN, the “invalid” floating-point exception is raised and a (canonical) quiet NaN (which should be the canonical version of that signaling NaN made quiet) is produced. For quiet NaN, infinity, and finite inputs, the functions raise no floating-point exceptions.

**F.10.9 Maximum, minimum, and positive difference functions**

**F.10.9.1 The fdim functions**

1 No additional requirements.

**F.10.9.2 The fmax functions**

1 If just one argument is a NaN, the `fmax` functions return the other argument (if both arguments are NaNs, the functions return a NaN).

2 The returned value is exact and is independent of the current rounding direction mode.

3 The body of the `fmax` function might be\(^{401}\)

```c
{
    double r = (isgreaterEqual(x, y) || isnan(y)) ? x : y;
    (void) canonicalize(&r, &r);
    return r;
}
```

\(^{400}\)As if `*x` * `1e0` were computed. Note also that this implementation does not handle signaling NaNs as required of implementations that define `FE_INEXACT_ALWAYS_SIGNAL`.

\(^{401}\)Ideally, `fmax` would be sensitive to the sign of zero, for example `fmax(-0.0, +0.0)` would return `+0.0`; however, implementation in software might be impractical.
F.10.9.3 The fmin functions
1 The fmin functions are analogous to the fmax functions (see F.10.9.2).
2 The returned value is exact and is independent of the current rounding direction mode.

F.10.9.4 The fmaxmag functions
1 If just one argument is a NaN, the fmaxmag functions return the other argument (if both arguments are NaNs, the functions return a NaN).
2 The returned value is exact and is independent of the current rounding direction mode.
3 The body of the fmaxmag function might be

```c
{ double r;
  double ax = fabs(x);
  double ay = fabs(y);
  if (isgreater(ax, ay)) (void)canonicalize(&r, &x);
  else if (isgreater(ay, ax)) (void)canonicalize(&r, &y);
  else r = fmax(x, y);
  return r;
}
```

F.10.9.5 The fminmag functions
1 The fminmag functions are analogous to the fmaxmag functions (F.10.9.4).
2 The returned value is exact and is independent of the current rounding direction mode.

F.10.10 Floating multiply-add
F.10.10.1 The fma functions
1 fma(x, y, z) computes xy + z, correctly rounded once.
2 fma(x, y, z) returns a NaN and optionally raises the “invalid” floating-point exception if one of x and y is infinite, the other is zero, and z is a NaN.
3 fma(x, y, z) returns a NaN and raises the “invalid” floating-point exception if one of x and y is infinite, the other is zero, and z is not a NaN.
4 fma(x, y, z) returns a NaN and raises the “invalid” floating-point exception if x times y is an exact infinity and z is also an infinity but with the opposite sign.

F.10.11 Functions that round result to narrower type
1 The functions that round their result to narrower type (7.12.14) are fully specified in IEC 60559. The returned value is dependent on the current rounding direction mode.
2 These functions treat zero and infinite arguments like the corresponding operation or function: +, -, *+, /, fma, or sqrt.

F.10.12 Total order functions
1 This subclause specifies the total order functions required by IEC 60559.
2 NOTE These functions are specified only in Annex F because they depend on details of IEC 60559 formats that might not be supported if __STDC_IEC_60559_BFP__ is not defined.

F.10.12.1 The totalorder functions

Synopsis
1

```c
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int totalorder(const double *x, const double *y);
int totalorderf(const float *x, const float *y);
```
int totalorderl(const long double *x, const long double *y);
#define __STDC_IEC_60559_DFP__
int totalorder32(const _Decimal32 *x, const _Decimal32 *y);
int totalorder64(const _Decimal64 *x, const _Decimal64 *y);
int totalorder128(const _Decimal128 *x, const _Decimal128 *y);
#undef __STDC_IEC_60559_DFP__

Description

The totalorder functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of *x, *y. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if *x or *y is a signaling NaN.

Returns

The totalorder functions return nonzero if and only if the total order relation is true for the ordered pair of *x, *y.

F.10.12.2 The totalordermag functions

Synopsis

#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int totalordermag(const double *x, const double *y);
int totalordermagf(const float *x, const float *y);
int totalordermagl(const long double *x, const long double *y);
#define __STDC_IEC_60559_DFP__
int totalordermagd32(const _Decimal32 *x, const _Decimal32 *y);
int totalordermagd64(const _Decimal64 *x, const _Decimal64 *y);
int totalordermagd128(const _Decimal128 *x, const _Decimal128 *y);
#undef __STDC_IEC_60559_DFP__

Description

The totalordermag functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of the magnitudes of *x, *y. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if *x or *y is a signaling NaN.

Returns

The totalordermag functions return nonzero if and only if the total order relation is true for the ordered pair of the magnitudes of *x, *y.

F.10.13 Payload functions

IEC 60559 defines the payload to be information contained in a quiet or signaling NaN. The payload is intended for implementation-defined diagnostic information about the NaN, such as where or how the NaN was created. The implementation interprets the payload as a non-negative integer suitable for use with the functions in this subclause, which get and set payloads. The implementation may restrict which payloads are admissible for the user to set.

NOTE These functions are specified only in Annex F because they depend on details of IEC 60559 formats that might not be supported if __STDC_IEC_60559_BFP__ is not defined.

F.10.13.1 The getpayload functions

Synopsis

#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
#define __STDC_IEC_60559_DFP__
Description

2 The `getpayload` functions extract the payload of a quiet or signaling NaN input and return it as a positive-signed floating-point integer. If `*x` is not a NaN, the return result is −1. These functions raise no floating-point exceptions, even if `*x` is a signaling NaN.

Returns

3 The `getpayload` functions return the payload of the NaN input as a positive-signed floating-point integer.

F.10.13.2 The `setpayload` functions

Synopsis

```c
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int setpayload(double *res, double pl);
int setpayloadf(float *res, float pl);
int setpayloadl(long double *res, long double pl);
#endif
```

Description

2 The `setpayload` functions create a quiet NaN with the payload specified by `pl` and a zero sign bit and store that NaN in the object pointed to by `*res`. If `pl` is not a floating-point integer representing an admissible payload, `*res` is set to +0.

Returns

3 If the `setpayload` functions stored the specified NaN, they return a zero value, otherwise a non-zero value (and `*res` is set to +0).

F.10.13.3 The `setpayloadsig` functions

Synopsis

```c
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int setpayloadsig(double *res, double pl);
int setpayloadsigf(float *res, float pl);
int setpayloadsigl(long double *res, long double pl);
#endif
```

Description

2 The `setpayloadsig` functions create a signaling NaN with the payload specified by `pl` and a zero sign bit and store that NaN in the object pointed to by `*res`. If `pl` is not a floating-point integer representing an admissible payload, `*res` is set to +0.

Returns

3 If the `setpayloadsig` functions stored the specified NaN, they return a zero value, otherwise a
non-zero value (and \*res is set to +0)."

F.10.14 Comparison macros

Relational operators and their corresponding comparison macros (7.4.2.17, 12.17) produce equivalent result values, even if argument values are represented in wider formats. Thus, comparison macro arguments represented in formats wider than their semantic types are not converted to the semantic types, unless the wide evaluation method converts operands of relational operators to their semantic types. The standard wide evaluation methods characterized by \texttt{FLT\_EVAL\_METHOD} equal to 1 or 2 (5.2.4.2.2), do not convert operands of relational operators to their semantic types.

F.10.14.1 The \texttt{iseqsig} macro

The \texttt{equality operator ==} and the \texttt{iseqsig} macro produce equivalent results, except that the \texttt{iseqsig} macro raises the "invalid" floating-point exception if an argument is a NaN.
Annex G
IEC 60559-compatible complex arithmetic

G.1 Introduction
This annex supplements Annex F to specify complex arithmetic for compatibility with IEC 60559 real floating-point arithmetic. An implementation that defines \texttt{__STDC_IEC_60559_COMPLEX__} or \texttt{__STDC_IEC_559_COMPLEX__} shall conform to the specifications in this annex.\footnote{Implementations that do not define \texttt{__STDC_IEC_60559_COMPLEX__} or \texttt{__STDC_IEC_559_COMPLEX__} are not required to conform to these specifications. The use of \texttt{__STDC_IEC_559_COMPLEX__} for this purpose is obsolescent and should be avoided in new code.}

G.2 Types

1. There is a new keyword \texttt{Imaginary}, which is used to specify imaginary types. It is used as a type specifier within declaration specifiers in the same way as \texttt{Complex} is (thus, \texttt{Imaginary float} is a valid type name).

2. There are three imaginary types, designated as \texttt{float Imaginary}, \texttt{double Imaginary}, and \texttt{long double Imaginary}. The imaginary types (along with the real floating and complex types) are floating types.

3. For imaginary types, the corresponding real type is given by deleting the keyword \texttt{Imaginary} from the type name.

4. Each imaginary type has the same representation and alignment requirements as the corresponding real type. The value of an object of imaginary type is the value of the real representation times the imaginary unit.

5. The imaginary type domain comprises the imaginary types.

G.3 Conventions

1. A complex or imaginary value with at least one infinite part is regarded as an infinity (even if its other part is a quiet NaN). A complex or imaginary value is a finite number if each of its parts is a finite number (neither infinite nor NaN). A complex or imaginary value is a zero if each of its parts is a zero.

G.4 Conversions

G.4.1 Imaginary types

Conversions among imaginary types follow rules analogous to those for real floating types.

G.4.2 Real and imaginary

1. When a value of imaginary type is converted to a real type other than \texttt{Bool},\footnote{See 6.3.1.2.} the result is a positive zero.

2. When a value of real type is converted to an imaginary type, the result is a positive imaginary zero.

G.4.3 Imaginary and complex

1. When a value of imaginary type is converted to a complex type, the real part of the complex result value is a positive zero and the imaginary part of the complex result value is determined by the conversion rules for the corresponding real types.

2. When a value of complex type is converted to an imaginary type, the real part of the complex value is discarded and the value of the imaginary part is converted according to the conversion rules for the corresponding real types.
G.5 Binary operators

1 The following subclauses supplement 6.5 in order to specify the type of the result for an operation with an imaginary operand.

2 For most operand types, the value of the result of a binary operator with an imaginary or complex operand is completely determined, with reference to real arithmetic, by the usual mathematical formula. For some operand types, the usual mathematical formula is problematic because of its treatment of infinities and because of undue overflow or underflow; in these cases the result satisfies certain properties (specified in G.5.1), but is not completely determined.

G.5.1 Multiplicative operators

Semantics

1 If one operand has real type and the other operand has imaginary type, then the result has imaginary type. If both operands have imaginary type, then the result has real type. (If either operand has complex type, then the result has complex type.)

2 If the operands are not both complex, then the result and floating-point exception behavior of the * operator is defined by the usual mathematical formula:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xu</td>
<td>i(xv)</td>
</tr>
<tr>
<td>iy</td>
<td>i(yu)</td>
<td>(-y)v</td>
</tr>
<tr>
<td>x + iy</td>
<td>(xu) + i(yu)</td>
<td>((-y)v) + i(xv)</td>
</tr>
</tbody>
</table>

3 If the second operand is not complex, then the result and floating-point exception behavior of the / operator is defined by the usual mathematical formula:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x/u</td>
<td>i((-x)/v)</td>
</tr>
<tr>
<td>iy</td>
<td>i(y/u)</td>
<td>y/v</td>
</tr>
<tr>
<td>x + iy</td>
<td>(x/u) + i(y/u)</td>
<td>i((-x)/v)</td>
</tr>
</tbody>
</table>

4 The * and / operators satisfy the following infinity properties for all real, imaginary, and complex operands:

   — if one operand is an infinity and the other operand is a nonzero finite number or an infinity, then the result of the * operator is an infinity;
   — if the first operand is an infinity and the second operand is a finite number, then the result of the / operator is an infinity;
   — if the first operand is a finite number and the second operand is an infinity, then the result of the / operator is a zero;
   — if the first operand is a nonzero finite number or an infinity and the second operand is a zero, then the result of the / operator is an infinity.

5 If both operands of the * operator are complex or if the second operand of the / operator is complex, the operator raises floating-point exceptions if appropriate for the calculation of the parts of the result, and may raise spurious floating-point exceptions.

6 EXAMPLE 1 Multiplication of double _Complex operands could be implemented as follows. Note that the imaginary unit I has imaginary type (see G.6).

```c
#include <math.h>
#include <complex.h>

/* Multiply z * w ...*/
double complex _Cmultd(double complex z, double complex w)
```

404 These properties are already implied for those cases covered in the tables, but are required for all cases (at least where the state for CX_LIMITED_RANGE is "off").
#pragma STDC FP_CONTRACT OFF
double a, b, c, d, ac, bd, ad, bc, x, y;
a = creal(z); b = cimag(z);
c = creal(w); d = cimag(w);
ac = a * c; bd = b * d;
ad = a * d; bc = b * c;
x = ac - bd; y = ad + bc;
if (isnan(x) && isnan(y)) {
    /* Recover infinities that computed as NaN+iNaN ... */
    int recalc = 0;
    if (isinf(a) || isinf(b)) { // z is infinite
        /* "Box" the infinity and change NaNs in the other factor to 0 */
        a = copysign(isinf(a) ? 1.0 : 0.0, a);
        b = copysign(isinf(b) ? 1.0 : 0.0, b);
        if (isnan(c)) c = copysign(0.0, c);
        if (isnan(d)) d = copysign(0.0, d);
        recalc = 1;
    }
    if (isinf(c) || isinf(d)) { // w is infinite
        /* "Box" the infinity and change NaNs in the other factor to 0 */
        c = copysign(isinf(c) ? 1.0 : 0.0, c);
        d = copysign(isinf(d) ? 1.0 : 0.0, d);
        if (isnan(a)) a = copysign(0.0, a);
        if (isnan(b)) b = copysign(0.0, b);
        recalc = 1;
    }
    if (!recalc && (isinf(ac) || isinf(bd) ||
                   isinf(ad) || isinf(bc))) {
        /* Recover infinities from overflow by changing NaNs to 0 ... */
        if (isnan(a)) a = copysign(0.0, a);
        if (isnan(b)) b = copysign(0.0, b);
        if (isnan(c)) c = copysign(0.0, c);
        if (isnan(d)) d = copysign(0.0, d);
        recalc = 1;
    }
    if (recalc) {
        x = INFINITY * (a + c - b + d);
        y = INFINITY * (a + d + b + c);
    }
}
return x + I * y;
}

7 This implementation achieves the required treatment of infinities at the cost of only one isnan test in ordinary (finite) cases. It is less than ideal in that undue overflow and underflow could occur.

8 EXAMPLE 2 Division of two double _Complex operands could be implemented as follows.

#include <math.h>
#include <complex.h>

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
    #pragma STDC FP_CONTRACT OFF
    double a, b, c, d, logbw, denom, x, y;
    int ilogbw = 0;
    a = creal(z); b = cimag(z);
    c = creal(w); d = cimag(w);
    logbw = logb(fmax(fabs(c), fabs(d)));
    if (isfinite(logbw)) {

§ G.5.1 IEC 60599-compatible complex arithmetic 473
ilogbw = (int)logbw;
c = scalbn(c, -ilogbw); d = scalbn(d, -ilogbw);
}
denom = c * c + d * d;
x = scalbn((a * c + b * d) / denom, -ilogbw);
y = scalbn((b * c - a * d) / denom, -ilogbw);

/* Recover infinities and zeros that computed as NaN+iNaN; */
/* the only cases are nonzero/zero, infinite/finite, and finite/infinite, ... */
if (isnan(x) && isnan(y)) {
  if ((denom == 0.0) && (!isnan(a) || !isnan(b))) {
    x = copysign(INFINITY, c) * a;
y = copysign(INFINITY, c) * b;
  } else if ((isinf(a) || isinf(b)) && isfinite(c) && isfinite(d)) {
    a = copysign(isinf(a) ? 1.0 : 0.0, a);
b = copysign(isinf(b) ? 1.0 : 0.0, b);
x = INFINITY * (a * c + b * d);
y = INFINITY * (b * c - a * d);
  } else if ((logbw == INFINITY) && isfinite(a) && isfinite(b)) {
    c = copysign(isinf(c) ? 1.0 : 0.0, c);
d = copysign(isinf(d) ? 1.0 : 0.0, d);
x = 0.0 * (a * c + b * d);
y = 0.0 * (b * c - a * d);
  }
  return x + I * y;
}

Scaling the denominator alleviates the main overflow and underflow problem, which is more serious than for multiplication. In the spirit of the multiplication example above, this code does not defend against overflow and underflow in the calculation of the numerator. Scaling with the `scalbn` function, instead of with division, provides better roundoff characteristics.

### G.5.2 Additive operators

#### Semantics
1. If both operands have imaginary type, then the result has imaginary type. (If one operand has real type and the other operand has imaginary type, or if either operand has complex type, then the result has complex type.)
2. In all cases the result and floating-point exception behavior of a `+` or `-` operator is defined by the usual mathematical formula:

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>iv</th>
<th>u + iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x±u</td>
<td>x±iv</td>
<td>(x±u)±iv</td>
</tr>
<tr>
<td>iy</td>
<td>±u + iy</td>
<td>i(y±v)</td>
<td>±u + i(y±v)</td>
</tr>
<tr>
<td>x + iy</td>
<td>(x±u) + iy</td>
<td>x + i(y±v)</td>
<td>(x±u) + i(y±v)</td>
</tr>
</tbody>
</table>

### G.6 Complex arithmetic `<complex.h>`

The macros `imaginary` and `_Imaginary_I`
are defined, respectively, as \texttt{.Imaginary} and a constant expression of type const float \texttt{.Imaginary} with the value of the imaginary unit. The macro

\begin{verbatim}
  I
\end{verbatim}

is defined to be \texttt{.Imaginary\_I} (not \texttt{.Complex\_I} as stated in 7.3). Notwithstanding the provisions of 7.1.3, a program may redefine and then perhaps redefine the macro \texttt{imaginary}.

This subclause contains specifications for the \texttt{<complex.h>} functions that are particularly suited to IEC 60559 implementations. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. Unless otherwise specified, where the symbol “±” occurs in both an argument and the result, the result has the same sign as the argument.

The functions are continuous onto both sides of their branch cuts, taking into account the sign of zero. For example, \texttt{csqrt(-2±i0)} = ±i\sqrt{2}.

Since complex and imaginary values are composed of real values, each function may be regarded as computing real values from real values. Except as noted, the functions treat real infinities, NaNs, signed zeros, subnormals, and the floating-point exception flags in a manner consistent with the IEC 60559 ones, in 7.3.9. These functions raise no floating-point exceptions.

In subsequent subclauses in G.6 “NaN” refers to a quiet NaN. The behavior of signaling NaNs in Annex G is implementation-defined.

The functions \texttt{cimag}, \texttt{conj}, \texttt{cproj}, and \texttt{creal} are fully specified for all implementations, including IEC 60559 ones, in 7.3.9. These functions raise no floating-point exceptions.

Each of the functions \texttt{cabs} and \texttt{carg} is specified by a formula in terms of a real function (whose special cases are covered in Annex F):

\begin{verbatim}
cabs(x + iy) = hypot(x, y) 
carg(x + iy) = atan2(y, x)
\end{verbatim}

Each of the functions \texttt{casin}, \texttt{catan}, \texttt{ccos}, \texttt{csin}, and \texttt{ctan} is specified implicitly by a formula in terms of other complex functions (whose special cases are specified below):

\begin{verbatim}
casin(z) = -i \texttt{casinh}(iz) 
catan(z) = -i \texttt{catanh}(iz) 
ccos(z) = \texttt{ccosh}(iz) 
csin(z) = -i \texttt{csinh}(iz) 
ctan(z) = -i \texttt{ctanh}(iz)
\end{verbatim}

For the other functions, the following subclauses specify behavior for special cases, including treatment of the “invalid” and “divide-by-zero” floating-point exceptions. For families of functions, the specifications apply to all of the functions even though only the principal function is shown. For a function \( f \) satisfying \( f(\text{conj}(z)) = \text{conj}(f(z)) \), the specifications for the upper half-plane imply the specifications for the lower half-plane; if the function \( f \) is also either even, \( f(-z) = f(z) \), or odd, \( f(-z) = -f(z) \), then the specifications for the first quadrant imply the specifications for the other three quadrants.

In the following subclauses, \texttt{cis}(y) is defined as \( \cos(y) + i \sin(y) \).

\section*{G.6.1 Trigonometric functions}

\subsection*{G.6.1.1 The \texttt{cacos} functions}

\begin{verbatim}
  — \texttt{cacos(\text{conj}(z)) = conj(cacos(z))}.
  — \texttt{cacos(±0 + i0) returns π/2 — i0}.
  — \texttt{cacos(±0 + iNaN) returns π/2 + iNaN}.
\end{verbatim}

\footnote{As noted in G.3, a complex value with at least one infinite part is regarded as an infinity even if its other part is a quiet NaN.}

\section*{§ G.6.1.1 IEC 60559-compatible complex arithmetic}
— \(\text{cacos}(x + i\infty)\) returns \(\pi/2 - i\infty\), for finite \(x\).

— \(\text{cacos}(x + iNaN)\) returns \(NaN + iNaN\) and optionally raises the “invalid” floating-point exception, for nonzero finite \(x\).

— \(\text{cacos}(-\infty + iy)\) returns \(-\infty\), for positive-signed finite \(y\).

— \(\text{cacos}(+\infty + iy)\) returns \(+0 - i\infty\), for positive-signed finite \(y\).

— \(\text{cacos}(-\infty + i\infty)\) returns \(3\pi/4 - i\infty\), for positive-signed finite \(y\).

— \(\text{cacos}(+\infty + i\infty)\) returns \(\pi/4 - i\infty\), for positive-signed finite \(y\).

— \(\text{cacos}(\pm\infty + iNaN)\) returns \(NaN\pm i\infty\) (where the sign of the imaginary part of the result is unspecified).

— \(\text{cacos}(NaN + iy)\) returns \(NaN + iNaN\) and optionally raises the “invalid” floating-point exception, for finite \(y\).

— \(\text{cacos}(NaN + i\infty)\) returns \(NaN - i\infty\).

— \(\text{cacos}(NaN + iNaN)\) returns \(NaN + iNaN\).

G.6.2 Hyperbolic functions

G.6.2.1 The \(\text{cacosh}\) functions

1

— \(\text{cacosh}(\text{conj}(z)) = \text{conj}(\text{cacosh}(z))\).

— \(\text{cacosh}(\pm 0 + i0)\) returns \(+0 + i\pi/2 \pm 0 + i\infty\).

— \(\text{cacosh}(x + i\infty)\) returns \(+\infty + i\pi/2 + \infty + i\infty\), for finite \(x\).

— \(\text{cacosh}(0 + iNaN)\) returns \(NaN \pm i\pi/2 + NaN \pm i\infty\) (where the sign of the imaginary part of the result is unspecified).

— \(\text{cacosh}(x + iNaN)\) returns \(NaN + iNaN\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(x\).

— \(\text{cacosh}(-\infty + iy)\) returns \(+\infty + i\pi\), for positive-signed finite \(y\).

— \(\text{cacosh}(+\infty + iy)\) returns \(+\infty + i0\), for positive-signed finite \(y\).

— \(\text{cacosh}(-\infty + i\infty)\) returns \(+\infty + i3\pi/4 + \infty + i\infty\).

— \(\text{cacosh}(+\infty + i\infty)\) returns \(+\infty + i\pi/4 + \infty + i\infty\).

— \(\text{cacosh}(\pm\infty + iNaN)\) returns \(+\infty + iNaN\).

— \(\text{cacosh}(NaN + iy)\) returns \(NaN + iNaN\) and optionally raises the “invalid” floating-point exception, for finite \(y\).

— \(\text{cacosh}(NaN + i\infty)\) returns \(+\infty + iNaN\).

— \(\text{cacosh}(NaN + iNaN)\) returns \(NaN + iNaN\).
G.6.2.2 The casinh functions

1. \( \text{casinh}(\text{conj}(z)) = \text{conj}((\text{casinh}(z)) \), and casinh is odd.

- casinh(0 + 0i) returns 0 + 0i.
- casinh(x + i\infty) returns \( +\infty + i\pi/2 + \infty + i\frac{\pi}{2} \) for positive-signed finite x.
- casinh(x + iNaN) returns NaN + iNaN and optionally raises the “invalid” floating-point exception, for finite x.

- casinh(\(+\infty + iy\)) returns \(+\infty + 0\) for positive-signed finite y.
- casinh(\(+\infty + i\infty\)) returns \(+\infty + i\pi/2 + \infty + i\frac{\pi}{2}\).
- casinh(\(+\infty + iNaN\)) returns \(+\infty + iNaN\).

- casinh(NaN + 0i) returns NaN + 0i.
- casinh(NaN + iy) returns NaN + iNaN and optionally raises the “invalid” floating-point exception, for finite nonzero y.
- casinh(NaN + i\infty) returns \( \pm\infty + iNaN \) (where the sign of the real part of the result is unspecified).

- casinh(NaN + iNaN) returns NaN + iNaN.

G.6.2.3 The catanh functions

1. \( \text{catanh}(\text{conj}(z)) = \text{conj}((\text{catanh}(z)) \), and catanh is odd.

- catanh(0 + 0i) returns 0 + 0i.
- catanh(0 + iNaN) returns 0 + iNaN.
- catanh(\(+1 + iy\)) returns \(+\infty + 0\) and raises the “divide-by-zero” floating-point exception.
- catanh(x + i\infty) returns \(+0 + i\pi/2 + 0 + i\frac{\pi}{2}\) for finite positive-signed x.
- catanh(x + iNaN) returns NaN + iNaN and optionally raises the “invalid” floating-point exception, for nonzero finite x.

- catanh(\(+\infty + iy\)) returns \(+0 + i\pi/2 + 0 + i\frac{\pi}{2}\) for finite positive-signed y.
- catanh(\(+\infty + i\infty\)) returns \(+0 + i\pi/2 + 0 + i\frac{\pi}{2}\).
- catanh(\(+\infty + iNaN\)) returns \(+0 + iNaN\).

- catanh(NaN + iy) returns NaN + iNaN and optionally raises the “invalid” floating-point exception, for finite y.
- catanh(NaN + i\infty) returns \( \pm0 + i\pi/2 + \pm0 + i\frac{\pi}{2} \) (where the sign of the real part of the result is unspecified).

- catanh(NaN + iNaN) returns NaN + iNaN.
G.6.2.4 The \texttt{ccosh} functions

1. \texttt{ccosh(conj(z)) = conj(ccosh(z))} and \texttt{ccosh} is even.

- \texttt{ccosh(+0 + i0)} returns \(1 + i0\).
- \texttt{ccosh(+0 + \infty)} returns \(\text{NaN} \pm i0\) (where the sign of the imaginary part of the result is unspecified) and raises the “invalid” floating-point exception.
- \texttt{ccosh(+0 + iNaN)} returns \(\text{NaN} \pm i0\) (where the sign of the imaginary part of the result is unspecified).
- \texttt{ccosh(x + i\infty)} returns \(\text{NaN} + i\text{NaN}\) and raises the “invalid” floating-point exception, for finite nonzero \(x\).
- \texttt{ccosh(x + iNaN)} returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(x\).
- \texttt{ccosh(+\infty + i0)} returns \(+\infty + i0\).
- \texttt{ccosh(+\infty + iy)} returns \(+\infty \circ\text{cis}(y)\), for finite nonzero \(y\).
- \texttt{ccosh(+\infty + i\infty)} returns \(\text{NaN} \pm i\text{NaN}\) (where the sign of the real part of the result is unspecified) and raises the “invalid” floating-point exception.
- \texttt{ccosh(+\infty + iNaN)} returns \(+\infty + i\text{NaN}\).
- \texttt{ccosh(\text{NaN} + i0)} returns \(\text{NaN} \pm i0\) (where the sign of the imaginary part of the result is unspecified).
- \texttt{ccosh(\text{NaN} + iy)} returns \(\text{NaN} \pm i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for all nonzero numbers \(y\).
- \texttt{ccosh(\text{NaN} + iNaN)} returns \(\text{NaN} \pm i\text{NaN}\).

G.6.2.5 The \texttt{csinh} functions

1. \texttt{csinh(conj(z)) = conj(csinh(z))}. and \texttt{csinh} is odd.

- \texttt{csinh(+0 + i0)} returns \(0 + i0\).
- \texttt{csinh(+0 + \infty)} returns \(\pm0 + i\text{NaN}\) (where the sign of the real part of the result is unspecified) and raises the “invalid” floating-point exception.
- \texttt{csinh(+0 + iNaN)} returns \(\pm0 + i\text{NaN}\) (where the sign of the real part of the result is unspecified).
- \texttt{csinh(x + i\infty)} returns \(\text{NaN} + i\text{NaN}\) and raises the “invalid” floating-point exception, for positive finite \(x\).
- \texttt{csinh(x + iNaN)} returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(x\).
- \texttt{csinh(+\infty + i0)} returns \(+\infty + i0\).
- \texttt{csinh(+\infty + iy)} returns \(+\infty \circ\text{cis}(y)\), for positive finite \(y\).
- \texttt{csinh(+\infty + i\infty)} returns \(\pm\infty + i\text{NaN}\) (where the sign of the real part of the result is unspecified) and raises the “invalid” floating-point exception.
- \texttt{csinh(+\infty + iNaN)} returns \(\pm\infty + i\text{NaN}\) (where the sign of the real part of the result is unspecified).
- \texttt{csinh(\text{NaN} + i0)} returns \(\text{NaN} + i0\).
- \texttt{csinh(\text{NaN} + iy)} returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for all nonzero numbers \(y\).
- \texttt{csinh(\text{NaN} + iNaN)} returns \(\text{NaN} + i\text{NaN}\).
G.6.2.6 The ctanh functions

1. \( \text{ctanh}(\text{conj}(z)) = \text{conj}(\text{ctanh}(z)) \) and \( \text{ctanh} \) is odd.
2. \( \text{ctanh}(+0 + i0) \) returns \(+0 + i0\).
3. \( \text{ctanh}(0 + i\infty) \) returns \(0 + i\text{NaN}\) and raises the “invalid” floating-point exception.
4. \( \text{ctanh}(x + i\infty) \) returns \(\text{NaN} + i\text{NaN}\) and raises the “invalid” floating-point exception, for finite nonzero \(x\).
5. \( \text{ctanh}(0 + i\text{NaN}) \) returns \(0 + i\text{NaN}\).
6. \( \text{ctanh}(x + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(x\).
7. \( \text{ctanh}(+\infty + i0) \) returns \(+\infty + i0\).
8. \( \text{ctanh}(\pm\infty + i\infty) \) returns \(\infty \pm i\text{NaN}\) and raises the “invalid” floating-point exception (where the sign of the real part of the result is unspecified).
9. \( \text{ctanh}(\pm\infty + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN} \pm i\text{NaN}\) (where the sign of the imaginary part of the result is unspecified).
10. \( \text{ctanh}(\text{NaN} + i0) \) returns \(\text{NaN} + i0\).
11. \( \text{ctanh}(\text{NaN} + iy) \) returns \(\text{NaN} + i\text{NaN} \pm i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(y\).

G.6.3 Exponential and logarithmic functions

G.6.3.1 The cexp functions

1. \( \text{cexp}(\text{conj}(z)) = \text{conj}(\text{cexp}(z)) \).
2. \( \text{cexp}(\pm0 + i0) \) returns \(1 + i0\).
3. \( \text{cexp}(x + i\infty) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite \(x\).
4. \( \text{cexp}(x + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite \(x\).
5. \( \text{cexp}(+\infty + i0) \) returns \(+\infty + i0\).
6. \( \text{cexp}(\pm\infty + iy) \) returns \(+0 \pm \text{cis}(y)\), for finite \(y\).
7. \( \text{cexp}(\pm\infty + iy) \) returns \(\pm0 \pm \text{cis}(y)\), for finite nonzero \(y\).
8. \( \text{cexp}(\pm\infty + i\infty) \) returns \(\pm\infty \pm i\text{NaN}\) (where the signs of the real and imaginary parts of the result are unspecified).
9. \( \text{cexp}(\pm\infty + i\text{NaN}) \) returns \(\pm\infty \pm i\text{NaN}\) (where the sign of the real part of the result is unspecified).
10. \( \text{cexp}(\text{NaN} + i\infty) \) returns \(\pm\infty + i\text{NaN}\) (where the sign of the real part of the result is unspecified).
11. \( \text{cexp}(\text{NaN} + iy) \) returns \(\text{NaN} + i\text{NaN} \pm i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite nonzero \(y\).
12. \( \text{cexp}(\text{NaN} + iy) \) returns \(\pm0 \pm i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for all nonzero numbers \(y\).
13. \( \text{cexp}(\text{NaN} + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\).
G.6.3.2 The clog functions

1. \( \text{clog}(\text{conj}(z)) = \text{conj}(\text{clog}(z)) \)

- \( \text{clog}(-0 + i0) \) returns \(-\infty + i\pi\) and raises the “divide-by-zero” floating-point exception.
- \( \text{clog}(+0 + i0) \) returns \(-\infty + i0\) and raises the “divide-by-zero” floating-point exception.
- \( \text{clog}(x + i\infty) \) returns \(\infty + i\pi/2 + \infty + i\pi/2\), for finite \(x\).
- \( \text{clog}(x + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite \(x\).
- \( \text{clog}(\infty + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\).

G.6.4 Power and absolute-value functions

G.6.4.1 The cpow functions

The \( \text{cpow} \) functions raise floating-point exceptions if appropriate for the calculation of the parts of the result, and may also raise spurious floating-point exceptions.\(^\text{406}\)

G.6.4.2 The csqrt functions

1. \( \text{csqrt}(\text{conj}(z)) = \text{conj}(\text{csqrt}(z)) \)

- \( \text{csqrt}(\pm0 + i0) \) returns \(+0 + i0\).
- \( \text{csqrt}(x + i\infty) \) returns \(+\infty + i\infty\), for all \(x\) (including \text{NaN}).
- \( \text{csqrt}(x + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite \(x\).
- \( \text{csqrt}(-\infty + iy) \) returns \(+0 + i\infty\), for finite positive-signed \(y\).
- \( \text{csqrt}(+\infty + iy) \) returns \(+\infty + i0\), for finite positive-signed \(y\).
- \( \text{csqrt}(-\infty + i\text{NaN}) \) returns \(\text{NaN}\pm i\infty\) (where the sign of the imaginary part of the result is unspecified).
- \( \text{csqrt}(+\infty + i\text{NaN}) \) returns \(+\infty + i\text{NaN}\).
- \( \text{csqrt}(\text{NaN} + iy) \) returns \(\text{NaN} + i\text{NaN}\) and optionally raises the “invalid” floating-point exception, for finite \(y\).
- \( \text{csqrt}(\text{NaN} + i\text{NaN}) \) returns \(\text{NaN} + i\text{NaN}\).

\(^\text{406}\)This allows \( \text{cpow}(z, c) \) to be implemented as \( \text{cexp}(c \text{clog}(z)) \) without precluding implementations that treat special cases more carefully.
G.7 Type-generic math <tgmath.h>

1 Type-generic macros that accept complex arguments also accept imaginary arguments. If an argument is imaginary, the macro expands to an expression whose type is real, imaginary, or complex, as appropriate for the particular function: if the argument is imaginary, then the types of `cos`, `cosh`, `fabs`, `carg`, `cimag`, and `creal` are real; the types of `sin`, `tan`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, and `atanh` are imaginary; and the types of the others are complex.

2 Given an imaginary argument, each of the type-generic macros `cos`, `sin`, `tan`, `cosh`, `sinh`, `tanh`, `asin`, `atan`, `asinh`, `atanh` is specified by a formula in terms of real functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cos(iy)</code></td>
<td><code>cosh(y)</code></td>
</tr>
<tr>
<td><code>sin(iy)</code></td>
<td><code>i sinh(y)</code></td>
</tr>
<tr>
<td><code>tan(iy)</code></td>
<td><code>i tanh(y)</code></td>
</tr>
<tr>
<td><code>cosh(iy)</code></td>
<td><code>cos(y)</code></td>
</tr>
<tr>
<td><code>sinh(iy)</code></td>
<td><code>i sin(y)</code></td>
</tr>
<tr>
<td><code>tanh(iy)</code></td>
<td><code>i tan(y)</code></td>
</tr>
<tr>
<td><code>asin(iy)</code></td>
<td><code>i asinh(y)</code></td>
</tr>
<tr>
<td><code>atan(iy)</code></td>
<td><code>i atanh(y)</code></td>
</tr>
<tr>
<td><code>asinh(iy)</code></td>
<td><code>i asin(y)</code></td>
</tr>
<tr>
<td><code>atanh(iy)</code></td>
<td><code>i atan(y)</code></td>
</tr>
</tbody>
</table>
Annex H
(informative)
Language independent arithmetic

H.1 Introduction
This annex documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1). LIA–1 is more general than IEC 60559 (Annex F) in that it covers integer and diverse floating-point arithmetics.

H.2 Types
The relevant C arithmetic types meet the requirements of LIA–1 types if an implementation adds notification of exceptional arithmetic operations and meets the 1 unit in the last place (ULP) accuracy requirement (LIA–1 subclause 5.2.8).

H.2.1 Boolean type
The LIA–1 data type Boolean is implemented by the C data type bool with values of true and false, all from <stdbool.h>.

H.2.2 Integer types
The signed C integer types int, long int, and the corresponding unsigned types are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values “integer_overflow” and “undefined”, then those types are LIA–1 conformant types. C’s unsigned integer types are “modulo” in the LIA–1 sense in that overflows or out-of-bounds results silently wrap. An implementation that defines signed integer types as also being modulo need not detect integer overflow, in which case, only integer divide-by-zero need be detected.

maxint INT_MAX, LONG_MAX, LLONG_MAX, UINT_MAX, ULONG_MAX, ULLONG_MAX
minint INT_MIN, LONG_MIN, LLONG_MIN

The parameter “bounded” is always true, and is not provided. The parameter “minint” is always 0 for the unsigned types, and is not provided for those types.

H.2.2.1 Integer operations
The integer operations on integer types are the following:

addl x + y
subl x - y
mul  x * y
divl, divtl x / y
reml, reml x % y
negl  -x

abs  abs(x), labs(x), llabs(x)

eql x == y
neql x != y
lsl  x < y
leql x <= y
\( gtr \quad x > y \)
\( geq \quad x \geq y \)

where \( x \) and \( y \) are expressions of the same integer type.

### H.2.3 Floating-point types

1. The C floating-point types \texttt{float}, \texttt{double}, and \texttt{long double} are compatible with LIA–1. If an implementation adds support for the LIA–1 exceptional values “underflow”, “floating_overflow”, and “undefined”, then those types are conformant with LIA–1. An implementation that uses IEC 60559 floating-point formats and operations (see Annex F) along with IEC 60559 status flags and traps has LIA–1 conformant types.

#### H.2.3.1 Floating-point parameters

1. The parameters for a floating-point data type can be accessed by the following:

   - \( r \) \texttt{FLT\_RADIX} 
   - \( p \) \texttt{FLT\_MANT\_DIG}, \texttt{DBL\_MANT\_DIG}, \texttt{LDBL\_MANT\_DIG} 
   - \( emax \) \texttt{FLT\_MAX\_EXP}, \texttt{DBL\_MAX\_EXP}, \texttt{LDBL\_MAX\_EXP} 
   - \( emin \) \texttt{FLT\_MIN\_EXP}, \texttt{DBL\_MIN\_EXP}, \texttt{LDBL\_MIN\_EXP} 

2. The derived constants for the floating-point types are accessed by the following:

   - \( fmax \) \texttt{FLT\_MAX}, \texttt{DBL\_MAX}, \texttt{LDBL\_MAX} 
   - \( fminN \) \texttt{FLT\_MIN}, \texttt{DBL\_MIN}, \texttt{LDBL\_MIN} 
   - \( epsilon \) \texttt{FLT\_EPSILON}, \texttt{DBL\_EPSILON}, \texttt{LDBL\_EPSILON} 
   - \( rnd\_style \) \texttt{FLT\_ROUNDS} 

#### H.2.3.2 Floating-point operations

1. The floating-point operations on floating-point types are the following:

   - \( addF \) \( x + y \) 
   - \( subF \) \( x - y \) 
   - \( mulF \) \( x \times y \) 
   - \( divF \) \( x / y \) 
   - \( negF \) \( -x \) 
   - \( absF \) \( \text{fabsf}(x), \text{fabs}(x), \text{fabsl}(x) \) 
   - \( exponentF \) \( 1.f+\logbf(x), 1.0+\logb(x), 1.L+\logbl(x) \) 
   - \( scaleF \) \( \text{scalbnf}(x, n), \text{scalbn}(x, n), \text{scalbnl}(x, n), \text{scalblnf}(x, li), \text{scalbln}(x, li), \text{scalblnl}(x, li) \) 
   - \( intpartF \) \( \text{modff}(x, &y), \text{modf}(x, &y), \text{modfl}(x, &y) \) 
   - \( fractpartF \) \( \text{modff}(x, &y), \text{modf}(x, &y), \text{modfl}(x, &y) \) 
   - \( eqF \) \( x == y \) 
   - \( neqF \) \( x != y \)
where \( x \) and \( y \) are expressions of the same floating-point type, \( n \) is of type \texttt{int}, and \( \texttt{li} \) is of type \texttt{long int}.

### H.2.3.3 Rounding styles

This document requires all floating types to use the same radix and rounding style, so that only one identifier for each is provided to map to LIA–1.

The \texttt{FLT_ROUNDS} parameter can be used to indicate the LIA–1 rounding styles:

- \texttt{truncate} if \( \texttt{FLT_ROUNDS} == 0 \)
- \texttt{nearest} if \( \texttt{FLT_ROUNDS} == 1 \)
- \texttt{other} if \( \texttt{FLT_ROUNDS} != 0 \) and \( \texttt{FLT_ROUNDS} != 1 \)

provided that an implementation extends \texttt{FLT_ROUNDS} to cover the rounding style used in all relevant LIA–1 operations, not just addition as in C.

### H.2.4 Type conversions

The LIA–1 type conversions are the following type casts:

- \( \texttt{cvt}\texttt{I'} \rightarrow \texttt{I} \): \( \texttt{(int)}\texttt{i}, \texttt{(long int)}\texttt{i}, \texttt{(long long int)}\texttt{i}, \texttt{(unsigned int)}\texttt{i}, \texttt{(unsigned long int)}\texttt{i}, \texttt{(unsigned long long int)}\texttt{i} \)
- \( \texttt{cvtF} \rightarrow \texttt{I} \): \( \texttt{(float)}\texttt{x}, \texttt{(double)}\texttt{x}, \texttt{(long double)}\texttt{x} \)
- \( \texttt{cvtI} \rightarrow \texttt{F} \): \( \texttt{(float)}\texttt{i}, \texttt{(double)}\texttt{i}, \texttt{(long double)}\texttt{i} \)
- \( \texttt{cvtF'} \rightarrow \texttt{F} \): \( \texttt{(float)}\texttt{x}, \texttt{(double)}\texttt{x}, \texttt{(long double)}\texttt{x} \)

In the above conversions from floating to integer, the use of \( \texttt{(cast)}\texttt{x} \) can be replaced with \( \texttt{(cast)}\texttt{round}(\texttt{x}), \texttt{(cast)}\texttt{rint}(\texttt{x}), \texttt{(cast)}\texttt{nearbyint}(\texttt{x}), \texttt{(cast)}\texttt{trunc}(\texttt{x}), \texttt{(cast)}\texttt{ceil}(\texttt{x}), \texttt{(cast)}\texttt{floor}(\texttt{x}) \). In addition, C’s floating-point to integer conversion functions, \texttt{rint()}\texttt{()}, \texttt{llrint()}\texttt{()}, \texttt{lround()}\texttt{()}, and \texttt{llround()}\texttt{()}, can be used. They all meet LIA–1’s requirements on floating to integer rounding for in-range values. For out-of-range values, the conversions shall silently wrap for the modulo types.

The \texttt{fmod()} function is useful for doing silent wrapping to unsigned integer types, e.g.,

\[
\texttt{fmod(fabs(rint(x)), 65536.0) or (0.0 <= (y = fmod(rint(x), 65536.0)) ? y : 65536.0 + y)}
\]

will compute an integer value in the range 0.0 to 65535.0 which can then be converted to \texttt{unsigned short int}. But, the \texttt{remainder()} function is not useful for doing silent wrapping to signed integer types, e.g., \texttt{remainder(rint(x), 65536.0)} will compute an integer value in the range \(-32767.0\) to \(+32768.0\) which is not, in general, in the range of \texttt{signed short int}.

C’s conversions (casts) from floating-point to floating-point can meet LIA–1 requirements if an implementation uses round-to-nearest (IEC 60559 default).

C’s conversions (casts) from integer to floating-point can meet LIA–1 requirements if an implementation uses round-to-nearest.

### H.3 Notification

Notification is the process by which a user or program is informed that an exceptional arithmetic operation has occurred. C’s operations are compatible with LIA–1 in that C allows an implementation to cause a notification to occur when any arithmetic operation returns an exceptional value as defined in LIA–1 clause 5.
H.3.1 Notification alternatives

1 LIA–1 requires at least the following two alternatives for handling of notifications: setting indicators or trap-and-terminate. LIA–1 allows a third alternative: trap-and-resume.

2 An implementation need only support a given notification alternative for the entire program. An implementation may support the ability to switch between notification alternatives during execution, but is not required to do so. An implementation can provide separate selection for each kind of notification, but this is not required.

3 C allows an implementation to provide notification. C’s SIGFPE (for traps) and FE_INVALID, FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW (for indicators) can provide LIA–1 notification.

4 C’s signal handlers are compatible with LIA–1. Default handling of SIGFPE can provide trap-and-terminate behavior, except for those LIA–1 operations implemented by math library function calls. User-provided signal handlers for SIGFPE allow for trap-and-resume behavior with the same constraint.

H.3.1.1 Indicators

1 C’s <fenv.h> status flags are compatible with the LIA–1 indicators.

2 The following mapping is for floating-point types:

<table>
<thead>
<tr>
<th>Status</th>
<th>LIA–1 Indicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>FE_INVALID, FE_DIVBYZERO</td>
</tr>
<tr>
<td>floating_overflow</td>
<td>FE_OVERFLOW</td>
</tr>
<tr>
<td>underflow</td>
<td>FE_UNDERFLOW</td>
</tr>
</tbody>
</table>

3 The floating-point indicator interrogation and manipulation operations are:

- set_indicators : feraiseexcept(i)
- clear_indicators : feclearexcept(i)
- test_indicators : fetestexcept(i)
- current_indicators : fetestexcept(FE_ALL_EXCEPT)

where i is an expression of type int representing a subset of the LIA–1 indicators.

4 C allows an implementation to provide the following LIA–1 required behavior: at program termination if any indicator is set the implementation shall send an unambiguous and “hard to ignore” message (see LIA–1 subclause 6.1.2)

5 LIA–1 does not make the distinction between floating-point and integer for “undefined”. This documentation makes that distinction because <fenv.h> covers only the floating-point indicators.

H.3.1.2 Traps

1 C is compatible with LIA–1’s trap requirements for arithmetic operations, but not for math library functions (which are not permitted to invoke a user’s signal handler for SIGFPE). An implementation can provide an alternative of notification through termination with a “hard-to-ignore” message (see LIA–1 subclause 6.1.3).

2 LIA–1 does not require that traps be precise.

3 C does require that SIGFPE be the signal corresponding to LIA–1 arithmetic exceptions, if there is any signal raised for them.

4 C supports signal handlers for SIGFPE and allows trapping of LIA–1 arithmetic exceptions. When LIA–1 arithmetic exceptions do trap, C’s signal-handler mechanism allows trap-and-terminate (either default implementation behavior or user replacement for it) or trap-and-resume, at the programmer’s option.
Annex I
(informative)

Common warnings

1 An implementation may generate warnings in many situations, none of which are specified as part of this document. The following are a few of the more common situations.

2 — A new struct or union type appears in a function prototype (6.2.1, 6.7.2.3).
   — A block with initialization of an object that has automatic storage duration is jumped into (6.2.4).
   — An implicit narrowing conversion is encountered, such as the assignment of a long int or a double to an int, or a pointer to void to a pointer to any type other than a character type (6.3).
   — A hexadecimal floating constant cannot be represented exactly in its evaluation format (6.4.4.2).
   — An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (6.4.4.4).
   — The characters /* are found in a comment (6.4.7).
   — An “unordered” binary operator (not comma, &, or | ) contains a side effect to an lvalue in one operand, and a side effect to, or an access to the value of, the identical lvalue in the other operand (6.5).
   — A function is called but no prototype has been supplied (6.5.2.2).
     The arguments in a function call do not agree in number and type with those of the parameters in a function definition that is not a prototype (6.5.2.2).— An object is defined but not used (6.7).
   — A value is given to an object of an enumerated type other than by assignment of an enumeration constant that is a member of that type, or an enumeration object that has the same type, or the value of a function that returns the same enumerated type (6.7.2.2).
   — An aggregate has a partly bracketed initialization (6.7.8).
   — A statement cannot be reached (6.8).
   — A statement with no apparent effect is encountered (6.8).
   — A constant expression is used as the controlling expression of a selection statement (6.8.4).
   — An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (6.10.1).
   — An unrecognized #pragma directive is encountered (6.10.6).
Annex J
(informative)
Portability issues

This annex collects some information about portability that appears in this document.

J.1 Unspecified behavior

The following are unspecified:

— The manner and timing of static initialization (5.1.2).
— The termination status returned to the hosted environment if the return type of `main` is not compatible with int (5.1.2.2.3).
— The values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).
— The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).
— The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).
— The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).
— The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).
— How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).
— Many aspects of the representations of types (6.2.6).
— The value of padding bytes when storing values in structures or unions (6.2.6.1).
— The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).
— The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).
— The values of any padding bits in integer representations (6.2.6.2).
— Whether certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (6.2.6.2).
— Whether two string literals result in distinct arrays (6.4.5).
— The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?;`, and comma operators (6.5).
— The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).
— The order of side effects among compound literal initialization list expressions (6.5.2.5).
— The order in which the operands of an assignment operator are evaluated (6.5.16).
— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).
— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

— Whether or not a size expression is evaluated when it is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator (6.7.6.2).

— The order in which any side effects occur among the initialization list expressions in an initializer (6.7.9).

— The layout of storage for function parameters (6.9.1).

— When a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token and the next preprocessing token from the source file is a `,` and the fully expanded replacement of that macro ends with the name of the first macro and the next preprocessing token from the source file is again a `,` whether that is considered a nested replacement (6.10.3).

— The order in which `#` and `##` operations are evaluated during macro substitution (6.10.3.2, 6.10.3.3).

— The line number of a preprocessing token, in particular `__LINE__`, that spans multiple physical lines (6.10.4).

— The line number of a preprocessing directive that spans multiple physical lines (6.10.4).

— The line number of a macro invocation that spans multiple physical or logical lines (6.10.4).

— The line number following a directive of the form `#line __LINE__ new-line` (6.10.4).

— The state of the floating-point status flags when execution passes from a part of the program translated with `FENV_ACCESS` "off" to a part translated with `FENV_ACCESS` "on" (7.6.1).

— The order in which `feraiseexcept` raises floating-point exceptions, except as stated in F.8.6 (7.6.4.3).

— Whether `math_errhandling` is a macro or an identifier with external linkage (7.12).

— The results of the `frexp` functions when the specified value is not a floating-point number (7.12.6.7).

— The numeric result of the `ilogb` functions when the correct value is outside the range of the return type (7.12.6.8, F10.3.8).

— The result of rounding when the value is out of range (7.12.9.5, 7.12.9.7, F10.6.5).

— The value stored by the `remquo` functions in the object pointed to by `quo` when `y` is zero (7.12.10.3).

— Whether a comparison macro argument that is represented in a format wider than its semantic type is converted to the semantic type (7.12.17).

— Whether `setjmp` is a macro or an identifier with external linkage (7.13).

— Whether `va_copy` and `va_end` are macros or identifiers with external linkage (7.16.1).

— The hexadecimal digit before the decimal point when a non-normalized floating-point number is printed with an `a` or `A` conversion specifier (7.21.6.1, 7.29.2.1).

— The value of the file position indicator after a successful call to the `ungetc` function for a text stream, or the `ungetwc` function for any stream, until all pushed-back characters are read or discarded (7.21.7.10, 7.29.3.10).

— The details of the value stored by the `fgetpos` function (7.21.9.1).
— The details of the value returned by the `ftell` function for a text stream (7.21.9.4).

— Whether the `strtod`, `strtof`, `strtol`, `wcstod`, `wcstof`, and `wcstold` functions convert a minus-signed sequence to a negative number directly or by negating the value resulting from converting the corresponding unsigned sequence (7.22.1.5, 7.29.4.1.1).

— The order and contiguity of storage allocated by successive calls to the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions (7.22.3).

— The amount of storage allocated by a successful call to the `calloc`, `malloc`, `realloc`, or `aligned_alloc` function when 0 bytes was requested (7.22.3).

— Whether a call to the `atexit` function that does not happen before the `exit` function is called will succeed (7.22.4.2).

— Whether a call to the `at_quick_exit` function that does not happen before the `quick_exit` function is called will succeed (7.22.4.3).

— Which of two elements that compare as equal is matched by the `bsearch` function (7.22.5.1).

— The order of two elements that compare as equal in an array sorted by the `qsort` function (7.22.5.2).

— The order in which destructors are invoked by `thrd_exit` (7.26.5.5).

— Whether calling `tss_delete` on a key while another thread is executing destructors affects the number of invocations of the destructors associated with the key on that thread (7.26.6.2).

— The encoding of the calendar time returned by the `time` function (7.27.2.4).

— The characters stored by the `strftime` or `wcsftime` function if any of the time values being converted is outside the normal range (7.27.3.5, 7.29.5.1).

— Whether an encoding error occurs if a `wchar_t` value that does not correspond to a member of the extended character set appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by `wcrtomb` (7.29.1).

— The conversion state after an encoding error occurs (7.29.6.3.2, 7.29.6.3.3, 7.29.6.4.1, 7.29.6.4.2, 7.29.6.4.3).

— The resulting value when the “invalid” floating-point exception is raised during IEC 60559 floating to integer conversion (F.4).

— Whether conversion of non-integer IEC 60559 floating values to integer raises the “inexact” floating-point exception (F.4).

— Whether or when library functions in `<math.h>` raise the “inexact” floating-point exception in an IEC 60559 conformant implementation (F.10).

— Whether or when library functions in `<math.h>` raise an undeserved “underflow” floating-point exception in an IEC 60559 conformant implementation (F.10).

— The exponent value stored by `frexp` for a NaN or infinity (F.10.3.7).

— The numeric result returned by the `lrint`, `llrint`, `lround`, and `llround` functions if the rounded value is outside the range of the return type (F.10.6.5, F.10.6.7).

— The sign of one part of the `complex` result of several math functions for certain special cases in IEC 60559 compatible implementations (G.6.1.1, G.6.2.2, G.6.2.3, G.6.2.4, G.6.2.5, G.6.2.6, G.6.3.1, G.6.4.2).
J.2 Undefined behavior

The behavior is undefined in the following circumstances:

- A “shall” or “shall not” requirement that appears outside of a constraint is violated (Clause 4).
- A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).
- Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).
- A program in a hosted environment does not define a function named `main` using one of the specified forms (5.1.2.2.1).
- The execution of a program contains a data race (5.1.2.4).
- A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).
- An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).
- The same identifier has both internal and external linkage in the same translation unit (6.2.2).
- An object is referred to outside of its lifetime (6.2.4).
- The value of a pointer to an object whose lifetime has ended is used (6.2.4).
- The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).
- A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).
- A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).
- The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).
- Two declarations of the same object or function specify types that are not compatible (6.2.7).
- A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).
- Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).
- Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).
- An lvalue does not designate an object when evaluated (6.3.2.1).
- A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).
- An lvalue designating an object of automatic storage duration that could have been declared with the `register` storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).
- An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).
— An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to `void`) is applied to a void expression (6.3.2.2).

— Conversion of a pointer to an integer type produces a value outside the range that can be represented (6.3.2.3).

— Conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3).

— A pointer is used to call a function whose type is not compatible with the referenced type (6.3.2.3).

— An unmatched ’ or ” character is encountered on a logical source line during tokenization (6.4).

— A reserved keyword token is used in translation phase 7 or 8 for some purpose other than as a keyword (6.4.1).

— A universal character name in an identifier does not designate a character whose encoding falls into one of the specified ranges (6.4.2.1).

— The initial character of an identifier is a universal character name designating a digit (6.4.2.1).

— Two identifiers differ only in nonsignificant characters (6.4.2.1).

— The identifier `__func__` is explicitly declared (6.4.2.2).

— The program attempts to modify a string literal (6.4.5).

— The characters ’, \", "/", or */ occur in the sequence between the < and > delimiters, or the characters ’, \", "/", or */ occur in the sequence between the " delimiters, in a header name preprocessing token (6.4.7).

— A side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object (6.5).

— An exceptional condition occurs during the evaluation of an expression (6.5).

— An object has its stored value accessed other than by an lvalue of an allowable type (6.5).

— For a call to a function without a function prototype in scope, the number of arguments does not equal the number of parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is defined with a function prototype, either the prototype ends with an ellipsis or the types of the arguments after default argument promotion are not compatible with the types of the parameters (6.5.2.2).

— For a call to a function without a function prototype in scope where the function is not defined with a function prototype, the types of the arguments after default argument promotion are not compatible with those of the parameters after promotion (with certain exceptions) (6.5.2.2). A function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function (6.5.2.2).

— A member of an atomic structure or union is accessed (6.5.2.3).

— The operand of the unary * operator has an invalid value (6.5.3.2).

— A pointer is converted to other than an integer or pointer type (6.5.4).

— The value of the second operand of the / or % operator is zero (6.5.5).

— If the quotient a/b is not representable, the behavior of both a/b and a%b (6.5.5).

— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
— Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

— Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

— An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression \(a[1][7]\) given the declaration `int a[4][5]`) (6.5.6).

— The result of subtracting two pointers is not representable in an object of type `ptrdiff_t` (6.5.6).

— An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

— An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would not be representable in the promoted type (6.5.7).

— Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

— An object is assigned to an inexactely overlapping object or to an exactly overlapping object with incompatible type (6.5.16.1).

— An expression that is required to be an integer constant expression does not have an integer type; has operands that are not integer constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, `_Alignof` expressions, or immediately-cast floating constants; or contains casts (outside operands to `sizeof` and `_Alignof` operators) other than conversions of arithmetic types to integer types (6.6).

— A constant expression in an initializer is not, or does not evaluate to, one of the following: an arithmetic constant expression, a null pointer constant, an address constant, or an address constant for a complete object type plus or minus an integer constant expression (6.6).

— An arithmetic constant expression does not have arithmetic type; has operands that are not integer constants, floating constants, enumeration constants, character constants, `sizeof` expressions whose results are integer constants, or `_Alignof` expressions; or contains casts (outside operands to `sizeof` or `_Alignof` operators) other than conversions of arithmetic types to arithmetic types (6.6).

— The value of an object is accessed by an array-subscript [], member-access . or ->, address &, or indirection * operator or a pointer cast in creating an address constant (6.6).

— An identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer (6.7).

— A function is declared at block scope with an explicit storage-class specifier other than `extern` (6.7.1).

— A structure or union is defined without any named members (including those specified indirectly via anonymous structures and unions) (6.7.2.1).

— An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array (6.7.2.1).

— When the complete type is needed, an incomplete structure or union type is not completed in the same scope by another declaration of the tag that defines the content (6.7.2.3).

— An attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type (6.7.3).
— An attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type (6.7.3).
— The specification of a function type includes any type qualifiers (6.7.3).
— Two qualified types that are required to be compatible do not have the identically qualified version of a compatible type (6.7.3).
— An object which has been modified is accessed through a restrict-qualified pointer to a const-qualified type, or through a restrict-qualified pointer and another pointer that are not both based on the same object (6.7.3.1).
— A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).
— A function with external linkage is declared with an **inline** function specifier, but is not also defined in the same translation unit (6.7.4).
— A function declared with a **_Noreturn** function specifier returns to its caller (6.7.4).
— The definition of an object has an alignment specifier and another declaration of that object has a different alignment specifier (6.7.5).
— Declarations of an object in different translation units have different alignment specifiers (6.7.5).
— Two pointer types that are required to be compatible are not identically qualified, or are not pointers to compatible types (6.7.6.1).
— The size expression in an array declaration is not a constant expression and evaluates at program execution time to a nonpositive value (6.7.6.2).
— In a context requiring two array types to be compatible, they do not have compatible element types, or their size specifiers evaluate to unequal values (6.7.6.2).
— A declaration of an array parameter includes the keyword **static** within the [ and ] and the corresponding argument does not provide access to the first element of an array with at least the specified number of elements (6.7.6.3).
— A storage-class specifier or type qualifier modifies the keyword **void** as a function parameter type list (6.7.6.3).
— In a context requiring two function types to be compatible, they do not have compatible return types, or their parameters disagree in use of the ellipses terminator or the number and type of parameters (after default argument promotion, when there is no parameter type list or when one type is specified by a function definition with an identifier list) (6.7.6.3).
— The value of an unnamed member of a structure or union is used (6.7.9).
— The initializer for a scalar is neither a single expression nor a single expression enclosed in braces (6.7.9).
— The initializer for a structure or union object that has automatic storage duration is neither an initializer list nor a single expression that has compatible structure or union type (6.7.9).
— The initializer for an aggregate or union, other than an array initialized by a string literal, is not a brace-enclosed list of initializers for its elements or members (6.7.9).
— An identifier with external linkage is used, but in the program there does not exist exactly one external definition for the identifier, or the identifier is not used and there exist multiple external definitions for the identifier (6.9).
A function definition includes an identifier list, but the types of the parameters are not declared in a following declaration list (6.9.1). An adjusted parameter type in a function definition is not a complete object type (6.9.1).

A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (6.9.1).

The } that terminates a function is reached, and the value of the function call is used by the caller (6.9.1).

An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (6.9.2).

A non-directive preprocessing directive is executed (6.10).

The token defined is generated during the expansion of a #if or #elif preprocessing directive, or the use of the defined unary operator does not match one of the two specified forms prior to macro replacement (6.10.1).

The #include preprocessing directive that results after expansion does not match one of the two header name forms (6.10.2).

The character sequence in an #include preprocessing directive does not start with a letter (6.10.2).

There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directives (6.10.3).

The result of the preprocessing operator # is not a valid character string literal (6.10.3.2).

The result of the preprocessing operator ## is not a valid preprocessing token (6.10.3.3).

The #line preprocessing directive that results after expansion does not match one of the two well-defined forms, or its digit sequence specifies zero or a number greater than 2147483647 (6.10.4).

A non-STDC #pragma preprocessing directive that is documented as causing translation failure or some other form of undefined behavior is encountered (6.10.6).

A #pragma STDC preprocessing directive does not match one of the well-defined forms (6.10.6).

The name of a predefined macro, or the identifier defined, is the subject of a #define or #undef preprocessing directive (6.10.8).

An attempt is made to copy an object to an overlapping object by use of a library function, other than as explicitly allowed (e.g., memmove) (Clause 7).

A file with the same name as one of the standard headers, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files (7.1.2).

A header is included within an external declaration or definition (7.1.2).

A function, object, type, or macro that is specified as being declared or defined by some standard header is used before any header that declares or defines it is included (7.1.2).

A standard header is included while a macro is defined with the same name as a keyword (7.1.2).

The program attempts to declare a library function itself, rather than via a standard header, but the declaration does not have external linkage (7.1.2).

The program declares or defines a reserved identifier, other than as allowed by 7.1.4 (7.1.3).

The program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3).
— An argument to a library function has an invalid value or a type not expected by a function with a variable number of arguments (7.1.4).

— The pointer passed to a library function array parameter does not have a value such that all address computations and object accesses are valid (7.1.4).

— The macro definition of `assert` is suppressed in order to access an actual function (7.2).

— The argument to the `assert` macro does not have a scalar type (7.2).

— The `CX_LIMITED_RANGE`, `FENV_ACCESS`, or `FP_CONTRACT` pragma is used in any context other than outside all external declarations or preceding all explicit declarations and statements inside a compound statement (7.3.4, 7.6.1, 7.12.2).

— The value of an argument to a character handling function is neither equal to the value of `EOF` nor representable as an `unsigned char` (7.4).

— A macro definition of `errno` is suppressed in order to access an actual object, or the program defines an identifier with the name `errno` (7.5).

— Part of the program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the `FENV_ACCESS` pragma “off” (7.6.1).

— The exception-mask argument for one of the functions that provide access to the floating-point status flags has a nonzero value not obtained by bitwise OR of the floating-point exception macros (7.6.4).

— The `fesetexceptflag` function is used to set floating-point status flags that were not specified in the call to the `fegetexceptflag` function that provided the value of the corresponding `fexcept_t` object (7.6.4.5).

— The argument to `fesetenv` or `feupdateenv` is neither an object set by a call to `fegetenv` or `feholdexcept`, nor is it an environment macro (7.6.6.3, 7.6.6.4).

— The value of the result of an integer arithmetic or conversion function cannot be represented (7.8.2.1, 7.8.2.2, 7.8.2.3, 7.8.2.4, 7.22.6.1, 7.22.6.2, 7.22.1).

— The program modifies the string pointed to by the value returned by the `setlocale` function (7.11.1.1).

— The program modifies the structure pointed to by the value returned by the `localeconv` function (7.11.2.1).

— A macro definition of `math_errhandling` is suppressed or the program defines an identifier with the name `math_errhandling` (7.12).

— An argument to a floating-point classification or comparison macro is not of real floating type (7.12.3, 7.12.17).

— A macro definition of `setjmp` is suppressed in order to access an actual function, or the program defines an external identifier with the name `setjmp` (7.13).

— An invocation of the `setjmp` macro occurs other than in an allowed context (7.13.2.1).

— The `longjmp` function is invoked to restore a nonexistent environment (7.13.2.1).

— After a `longjmp`, there is an attempt to access the value of an object of automatic storage duration that does not have volatile-qualified type, local to the function containing the invocation of the corresponding `setjmp` macro, that was changed between the `setjmp` invocation and `longjmp` call (7.13.2.1).

— The program specifies an invalid pointer to a signal handler function (7.14.1.1).
— A signal handler returns when the signal corresponded to a computational exception (7.14.1.1).

— A signal handler called in response to SIGFPE, SIGILL, SIGSEGV, or any other implementation-defined value corresponding to a computational exception returns (7.14.1.1).

— A signal occurs as the result of calling the `abort` or `raise` function, and the signal handler calls the `raise` function (7.14.1.1).

— A signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler refers to an object with static or thread storage duration that is not a lock-free atomic object other than by assigning a value to an object declared as `volatile sig_atomic_t`, or calls any function in the standard library other than the `abort` function, the `_Exit` function, the `quick_exit` function, the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free, the `atomic_is_lock_free` function with any atomic argument, or the `signal` function (for the same signal number) (7.14.1.1).

— The value of `errno` is referred to after a signal occurred other than as the result of calling the `abort` or `raise` function and the corresponding signal handler obtained a `SIG_ERR` return from a call to the `signal` function (7.14.1.1).

— A signal is generated by an asynchronous signal handler (7.14.1.1).

— The `signal` function is used in a multi-threaded program (7.14.1.1).

— A function with a variable number of arguments attempts to access its varying arguments other than through a properly declared and initialized `va_list` object, or before the `va_start` macro is invoked (7.16, 7.16.1.1, 7.16.1.4).

— The macro `va_arg` is invoked using the parameter `ap` that was passed to a function that invoked the macro `va_arg` with the same parameter (7.16).

— A macro definition of `va_start`, `va_arg`, `va_copy`, or `va_end` is suppressed in order to access an actual function, or the program defines an external identifier with the name `va_copy` or `va_end` (7.16.1).

— The `va_start` or `va_copy` macro is invoked without a corresponding invocation of the `va_end` macro in the same function, or vice versa (7.16.1, 7.16.1.2, 7.16.1.3, 7.16.1.4).

— The type parameter to the `va_arg` macro is not such that a pointer to an object of that type can be obtained simply by postfixing a `*` (7.16.1.1).

— The `va_arg` macro is invoked when there is no actual next argument, or with a specified type that is not compatible with the promoted type of the actual next argument, with certain exceptions (7.16.1.1).

— The `va_copy` or `va_start` macro is called to initialize a `va_list` that was previously initialized by either macro without an intervening invocation of the `va_end` macro for the same `va_list` (7.16.1.2, 7.16.1.4).

— The parameter `parmN` of a `va_start` macro is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions (7.16.1.4).

— The macro definition of a generic function is suppressed in order to access an actual function (7.17.1).

— The `type parameter of an offsetof macro defines a new type (7.19).`

— The `member-designator` parameter of an `offsetof` macro is an invalid right operand of the `.` operator for the `type` parameter, or designates a bit-field (7.19).

— The argument in an instance of one of the integer-constant macros is not a decimal, octal, or hexadecimal constant, or it has a value that exceeds the limits for the corresponding type (7.20.4).
— A byte input/output function is applied to a wide-oriented stream, or a wide character input/output function is applied to a byte-oriented stream (7.21.2).

— Use is made of any portion of a file beyond the most recent wide character written to a wide-oriented stream (7.21.2).

— The value of a pointer to a `FILE` object is used after the associated file is closed (7.21.3).

— The stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input (7.21.5.2).

— The string pointed to by the `mode` argument in a call to the `fopen` function does not exactly match one of the specified character sequences (7.21.5.3).

— An output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation with an intervening call to a file positioning function (7.21.5.3).

— An attempt is made to use the contents of the array that was supplied in a call to the `setvbuf` function (7.21.5.6).

— There are insufficient arguments for the format in a call to one of the formatted input/output functions, or an argument does not have an appropriate type (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— The format in a call to one of the formatted input/output functions or to the `strftime` or `wcsftime` function is not a valid multibyte character sequence that begins and ends in its initial shift state (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— In a call to one of the formatted output functions, a precision appears with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses an asterisk to denote an argument-supplied field width or precision, but the corresponding argument is not provided (7.21.6.1, 7.29.2.1).

— A conversion specification for a formatted output function uses a `#` or `0` flag with a conversion specifier other than those described (7.21.6.1, 7.29.2.1).

— A conversion specification for one of the formatted input/output functions uses a length modifier with a conversion specifier other than those described (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An `s` conversion specifier is encountered by one of the formatted output functions, and the argument is missing the null terminator (unless a precision is specified that does not require null termination) (7.21.6.1, 7.29.2.1).

— An `n` conversion specification for one of the formatted input/output functions includes any flags, an assignment-suppressing character, a field width, or a precision (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— A `%` conversion specifier is encountered by one of the formatted input/output functions, but the complete conversion specification is not exactly `%` (7.21.6.1, 7.21.6.2, 7.29.2.1, 7.29.2.2).

— An invalid conversion specification is found in the format for one of the formatted input/output functions, or the `strftime` or `wcsftime` function (7.21.6.1, 7.21.6.2, 7.27.3.5, 7.29.2.1, 7.29.2.2, 7.29.5.1).

— The number of characters or wide characters transmitted by a formatted output function (or written to an array, or that would have been written to an array) is greater than `INT_MAX` (7.21.6.1, 7.29.2.1).
— The number of input items assigned by a formatted input function is greater than \texttt{INT\_MAX} (7.21.6.2, 7.29.2.2).

— The result of a conversion by one of the formatted input functions cannot be represented in the corresponding object, or the receiving object does not have an appropriate type (7.21.6.2, 7.29.2.2).

— A \texttt{c}, \texttt{s}, or \texttt{[} conversion specifier is encountered by one of the formatted input functions, and the array pointed to by the corresponding argument is not large enough to accept the input sequence (and a null terminator if the conversion specifier is \texttt{s} or \texttt{[}) (7.21.6.2, 7.29.2.2).

— A \texttt{c}, \texttt{s}, or \texttt{[} conversion specifier with an \texttt{l} qualifier is encountered by one of the formatted input functions, but the input is not a valid multibyte character sequence that begins in the initial shift state (7.21.6.2, 7.29.2.2).

— The input item for a \texttt{\%p} conversion by one of the formatted input functions is not a value converted earlier during the same program execution (7.21.6.2, 7.29.2.2).

— The \texttt{vfprintf}, \texttt{vfscanf}, \texttt{vprintf}, \texttt{vscanf}, \texttt{vsnprintf}, \texttt{vsprintf}, \texttt{vsscanf}, \texttt{vfwprintf}, \texttt{vfwscanf}, \texttt{vswprintf}, \texttt{vswscanf}, \texttt{vwprintf}, or \texttt{vwscanf} function is called with an improperly initialized \texttt{va\_list} argument, or the argument is used (other than in an invocation of \texttt{va\_end}) after the function returns (7.21.6.8, 7.21.6.9, 7.21.6.10, 7.21.6.11, 7.21.6.12, 7.21.6.13, 7.21.6.14, 7.29.2.5, 7.29.2.6, 7.29.2.7, 7.29.2.8, 7.29.2.9, 7.29.2.10).

— The contents of the array supplied in a call to the \texttt{fgets} or \texttt{fgetws} function are used after a read error occurred (7.21.7.2, 7.29.3.2).

— The file position indicator for a binary stream is used after a call to the \texttt{ungetc} function where its value was zero before the call (7.21.7.10).

— The file position indicator for a stream is used after an error occurred during a call to the \texttt{fread} or \texttt{fwrite} function (7.21.8.1, 7.21.8.2).

— A partial element read by a call to the \texttt{fread} function is used (7.21.8.1).

— The \texttt{fseek} function is called for a text stream with a nonzero offset and either the offset was not returned by a previous successful call to the \texttt{ftell} function on a stream associated with the same file or \texttt{whence} is not \texttt{SEEK\_SET} (7.21.9.2).

— The \texttt{fsetpos} function is called to set a position that was not returned by a previous successful call to the \texttt{fgetpos} function on a stream associated with the same file (7.21.9.3).

— A non-null pointer returned by a call to the \texttt{calloc}, \texttt{malloc}, \texttt{realloc}, or \texttt{aligned\_alloc} function with a zero requested size is used to access an object (7.22.3).

— The value of a pointer that refers to space deallocated by a call to the \texttt{free} or \texttt{realloc} function is used (7.22.3).

— The pointer argument to the \texttt{free} or \texttt{realloc} function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to \texttt{free} or \texttt{realloc} (7.22.3.3, 7.22.3.5).

— The value of the object allocated by the \texttt{malloc} function is used (7.22.3.4).

— The values of any bytes in a new object allocated by the \texttt{realloc} function beyond the size of the old object are used (7.22.3.5).

— The program calls the \texttt{exit} or \texttt{quick\_exit} function more than once, or calls both functions (7.22.4.4, 7.22.4.7).

— During the call to a function registered with the \texttt{atexit} or \texttt{at\_quick\_exit} function, a call is made to the \texttt{longjmp} function that would terminate the call to the registered function (7.22.4.4, 7.22.4.7).
— The string set up by the `getenv` or `strerror` function is modified by the program (7.22.4.6, 7.24.6.2).

— A signal is raised while the `quick_exit` function is executing (7.22.4.7).

— A command is executed through the `system` function in a way that is documented as causing termination or some other form of undefined behavior (7.22.4.8).

— A searching or sorting utility function is called with an invalid pointer argument, even if the number of elements is zero (7.22.5).

— The comparison function called by a searching or sorting utility function alters the contents of the array being searched or sorted, or returns ordering values inconsistently (7.22.5).

— The array being searched by the `bsearch` function does not have its elements in proper order (7.22.5.1).

— The current conversion state is used by a multibyte/wide character conversion function after changing the `LC_CTYPE` category (7.22.7).

— A string or wide string utility function is instructed to access an array beyond the end of an object (7.24.1, 7.29.4).

— A string or wide string utility function is called with an invalid pointer argument, even if the length is zero (7.24.1, 7.29.4).

— The contents of the destination array are used after a call to the `strxfrm, strftime, wcsxfrm, or wcstok` function in which the specified length was too small to hold the entire null-terminated result (7.24.4.5, 7.27.3.5, 7.29.4.4.4, 7.29.5.1).

— The first argument in the very first call to the `strtok` or `wcstok` is a null pointer (7.24.5.8, 7.29.4.5.7).

— The type of an argument to a type-generic macro is not compatible with the type of the corresponding parameter of the selected function (7.25).

— Arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type (7.25).

— Arguments for generic parameters of a type-generic macro are such that neither `<math.h>` and `<complex.h>` define a function whose generic parameters have the determined corresponding real type (7.25).

— A complex argument is supplied for a generic parameter of a type-generic macro that has no corresponding complex function (7.25).

— A decimal floating argument is supplied for a generic parameter of a type-generic macro that expects a complex argument (7.25).

— A standard floating or complex argument is supplied for a generic parameter of a type-generic macro that expects a decimal floating type argument (7.25).

— A non-recursive mutex passed to `mtx_lock` is locked by the calling thread (7.26.4.3).

— The mutex passed to `mtx_timedlock` does not support timeout (7.26.4.4).

— The mutex passed to `mtx_unlock` is not locked by the calling thread (7.26.4.6).

— The thread passed to `thrd_detach` or `thrd_join` was previously detached or joined with another thread (7.26.5.3, 7.26.5.6).

— The `tss_create` function is called from within a destructor (7.26.6.1).
— The key passed to `tss_delete`, `tss_get`, or `tss_set` was not returned by a call to `tss_create` before the thread commenced executing destructors (7.26.6.2, 7.26.6.3, 7.26.6.4).

— At least one member of the broken-down time passed to `asctime` contains a value outside its normal range, or the calculated year exceeds four digits or is less than the year 1000 (7.27.3.1).

— The argument corresponding to an `s` specifier without an `l` qualifier in a call to the `fwprintf` function does not point to a valid multibyte character sequence that begins in the initial shift state (7.29.2.11).

— In a call to the `wcstok` function, the object pointed to by `ptr` does not have the value stored by the previous call for the same wide string (7.29.4.5.7).

— An `mbstate_t` object is used inappropriately (7.29.6).

— The value of an argument of type `wint_t` to a wide character classification or case mapping function is neither equal to the value of `WEOF` nor representable as a `wchar_t` (7.30.1).

— The `iswctype` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctype` function that returned the description (7.30.2.2.1).

— The `towctrans` function is called using a different `LC_CTYPE` category from the one in effect for the call to the `wctrans` function that returned the description (7.30.3.2.1).

### J.3 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

#### J.3.1 Translation

— How a diagnostic is identified (3.10, 5.1.1.3).

— Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

#### J.3.2 Environment

— The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

— The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

— The effect of program termination in a freestanding environment (5.1.2.1).

— An alternative manner in which the `main` function may be defined (5.1.2.2.1).

— The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

— What constitutes an interactive device (5.1.2.3).

— Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

— The set of signals, their semantics, and their default handling (7.14).

— Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

— Signals for which the equivalent of `signal(sig, SIG_IGN)`; is executed at program startup (7.14.1.1).

— The set of environment names and the method for altering the environment list used by the `getenv` function (7.22.4.6).

— The manner of execution of the string by the `system` function (7.22.4.8).
J.3.3 Identifiers

1. Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

2. The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

J.3.4 Characters

1. The number of bits in a byte (3.6).

2. The values of the members of the execution character set (5.2.1).

3. The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

4. The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).

5. Which of signed char or unsigned char has the same range, representation, and behavior as “plain” char (6.2.5, 6.3.1.1).

6. The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

7. The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

8. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

9. The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

10. Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

11. The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

12. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

13. The encoding of any of wchar_t, char16_t, and char32_t where the corresponding standard encoding macro (__STDC_ISO_10646__, __STDC_UTF_16__, or __STDC_UTF_32__) is not defined (6.10.8.2).

J.3.5 Integers

1. Any extended integer types that exist in the implementation (6.2.5).

2. Whether signed integer types are represented using sign and magnitude, two’s complement, or ones’ complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

3. The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

4. The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

5. The results of some bitwise operations on signed integers (6.5).
J.3.6 Floating point

— The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

— The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

— The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

— The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

— The evaluation methods characterized by non-standard negative values of `DEC_EVAL_METHOD` (5.2.4.2.3).

— If decimal floating types are supported (6.2.5).

— The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

— The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

— How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

— Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

— The default state for the `FENV_ACCESS` pragma (7.6.1).

— Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

— The default state for the `FP_CONTRACT` pragma (7.12.2).

J.3.7 Arrays and pointers

— The result of converting a pointer to an integer or vice versa (6.3.2.3).

— The size of the result of subtracting two pointers to elements of the same array (6.5.6).

J.3.8 Hints

— The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).

— The extent to which suggestions made by using the `inline` function specifier are effective (6.7.4).

J.3.9 Structures, unions, enumerations, and bit-fields

— Whether a “plain” `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.7.2, 6.7.2.1).

— Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1).

— Whether atomic types are permitted for bit-fields (6.7.2.1).

— Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

— The order of allocation of bit-fields within a unit (6.7.2.1).

— The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

— The integer type compatible with each enumerated type (6.7.2.2).
J.3.10 Qualifiers

— What constitutes an access to an object that has volatile-qualified type (6.7.3).

J.3.11 Preprocessing directives

— The locations within #pragma directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

— How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

— Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

— Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

— The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

— How the named source file is searched for in an included " " delimited header (6.10.2).

— The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).

— The nesting limit for #include processing (6.10.2).

— Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

— The behavior on each recognized non-STD #pragma directive (6.10.6).

— The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8.1).

J.3.12 Library functions

— Any library facilities available to a freestanding program, other than the minimal set required by Clause 4 (5.1.2.1).

— The format of the diagnostic printed by the assert macro (7.2.1.1).

— The representation of the floating-point status flags stored by the fegetexceptflag function (7.6.4.2).

— Whether the feraiseexcept function raises the “inexact” floating-point exception in addition to the “overflow” or “underflow” floating-point exception (7.6.4.3).

— Strings other than “C” and “ ” that may be passed as the second argument to the setlocale function (7.11.1.1).

— The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than 0 (7.12).

— The types defined for _Decimal32_t and _Decimal64_t when the value of the _DEC_EVAL_METHOD macro is less than 0 (7.12).

— Domain errors for the mathematics functions, other than those required by this document (7.12.1).

— The values returned by the mathematics functions on domain errors or pole errors (7.12.1).
The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the “underflow” floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

Whether a domain error occurs or zero is returned when a `remainder` function has a second argument of zero (7.12.10.2).

The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

The byte order of decimal floating type encodings (7.12.16).

Whether a domain error occurs or zero is returned when a `remquo` function has a second argument of zero (7.12.10.3).

Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

The null pointer constant to which the macro `NULL` expands (7.19).

Whether the last line of a text stream requires a terminating new-line character (7.21.2).

Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

The number of null characters that may be appended to data written to a binary stream (7.21.2).

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

The characteristics of file buffering (7.21.3).

Whether a zero-length file actually exists (7.21.3).

The rules for composing valid file names (7.21.3).

Whether the same file can be simultaneously open multiple times (7.21.3).

The nature and choice of encodings used for multibyte characters in files (7.21.3).

The effect of the `remove` function on an open file (7.21.4.1).

The effect if a file with the new name exists prior to a call to the `rename` function (7.21.4.2).

Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

The interpretation of a `^-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%{ conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).
— The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

— The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.22.1.5, 7.29.4.1.1).

— Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.5, 7.29.4.1.1).

— The meaning of any d-char or d-wchar sequence in a string representing a NaN that is converted by the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function (7.22.1.6, 7.29.4.1.2).

— Whether or not the `strtod32`, `strtod64`, `strtod128`, `wcstod32`, `wcstod64`, or `wcstod128` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.6, 7.29.4.1.2).

— Whether the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

— Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.22.4.1, 7.22.4.5).

— The termination status returned to the host environment by the `abort`, `exit`, `_Exit`, or `quick_exit` function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

— The range and precision of times representable in `clock_t` and `time_t` (7.27).

— The local time zone and Daylight Saving Time (7.27.1).

— The era for the `clock` function (7.27.2.1).

— The `TIME_UTC` epoch (7.27.2.5).

— The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the "C" locale (7.27.3.5, 7.29.5.1).

— Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

### J.3.13 Architecture

1 — The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.20.2, 7.20.3).

— The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

— The number, order, and encoding of bytes in any object (when not explicitly specified in this document) (6.2.6.1).

— Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

— Valid alignment values other than those returned by an `_Alignof` expression for fundamental types, if any (6.2.8).

— The value of the result of the `sizeof` and `_Alignof` operators (6.5.3.4).
J.4 Locale-specific behavior

The following characteristics of a hosted environment are locale-specific and are required to be documented by the implementation:

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).
- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).
- The shift states used for the encoding of multibyte characters (5.2.1.2).
- The direction of writing of successive printing characters (5.2.2).
- The decimal-point character (7.1.1).
- The set of printing characters (7.4, 7.30.2).
- The set of control characters (7.4, 7.30.2).
- The sets of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).
- The native environment (7.11.1.1).
- Additional subject sequences accepted by the numeric conversion functions (7.22.1, 7.29.4.1).
- The collation sequence of the execution character set (7.24.4.3, 7.29.4.4.2).
- The contents of the error message strings set up by the `strerror` function (7.24.6.2).
- The formats for time and date (7.27.3.5, 7.29.5.1).
- Character mappings that are supported by the `towctrans` function (7.30.1).
- Character classifications that are supported by the `iswctype` function (7.30.1).

J.5 Common extensions

The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, extra library functions declared in standard headers, or predefined macros with names that do not begin with an underscore.

J.5.1 Environment arguments

In a hosted environment, the `main` function receives a third argument, `char *envp[]`, that points to a null-terminated array of pointers to `char`, each of which points to a string that provides information about the environment for this execution of the program (5.1.2.2.1).

J.5.2 Specialized identifiers

Characters other than the underscore `_`, letters, and digits, that are not part of the basic source character set (such as the dollar sign `$`, or characters in national character sets) may appear in an identifier (6.4.2).

J.5.3 Lengths and cases of identifiers

All characters in identifiers (with or without external linkage) are significant (6.4.2).
J.5.4 Scopes of identifiers
1 A function identifier, or the identifier of an object the declaration of which contains the keyword `extern`, has file scope (6.2.1).

J.5.5Writable string literals
1 String literals are modifiable (in which case, identical string literals should denote distinct objects) (6.4.5).

J.5.6 Other arithmetic types
1 Additional arithmetic types, such as `__int128` or `double double`, and their appropriate conversions are defined (6.2.5, 6.3.1). Additional floating types may have more range or precision than `long double`, may be used for evaluating expressions of other floating types, and may be used to define `float_t` or `double_t`. Additional floating types may also have less range or precision than `float`.

J.5.7 Function pointer casts
1 A pointer to an object or to `void` may be cast to a pointer to a function, allowing data to be invoked as a function (6.5.4).
2 A pointer to a function may be cast to a pointer to an object or to `void`, allowing a function to be inspected or modified (for example, by a debugger) (6.5.4).

J.5.8 Extended bit-field types
1 A bit-field may be declared with a type other than `__Bool`, `unsigned int`, or `signed int`, with an appropriate maximum width (6.7.2.1).

J.5.9 The `fortran` keyword
1 The `fortran` function specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that a different representation for the external name is to be generated (6.7.4).

J.5.10 The `asm` keyword
1 The `asm` keyword may be used to insert assembly language directly into the translator output (6.8). The most common implementation is via a statement of the form:

```
asm (character-string-literal);
```

J.5.11 Multiple external definitions
1 There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword `extern`; if the definitions disagree, or more than one is initialized, the behavior is undefined (6.9.2).

J.5.12 Predefined macro names
1 Macro names that do not begin with an underscore, describing the translation and execution environments, are defined by the implementation before translation begins (6.10.8).

J.5.13 Floating-point status flags
1 If any floating-point status flags are set on normal termination after all calls to functions registered by the `atexit` function have been made (see 7.22.4.4), the implementation writes some diagnostics indicating the fact to the `stderr` stream, if it is still open.

J.5.14 Extra arguments for signal handlers
1 Handlers for specific signals are called with extra arguments in addition to the signal number (7.14.1.1).


J.5.15 Additional stream types and file-opening modes
1 Additional mappings from files to streams are supported (7.21.2).
2 Additional file-opening modes may be specified by characters appended to the mode argument of the fopen function (7.21.5.3).

J.5.16 Defined file position indicator
1 The file position indicator is decremented by each successful call to the ungetc or ungetwc function for a text stream, except if its value was zero before a call (7.21.7.10, 7.29.3.10).

J.5.17 Math error reporting
1 Functions declared in <complex.h> and <math.h> raise SIGFPE to report errors instead of, or in addition to, setting errno or raising floating-point exceptions (7.3, 7.12).

J.6 Reserved identifiers and keywords
1 A lot of identifier preprocessing tokens are used for specific purposes in regular clauses or appendices from translation phase 3 onwards. Using any of these for a purpose different from their description in this document, even if the use is in a context where they are normatively permitted, may have an impact on the portability of code and should thus be avoided.

J.6.1 Rule based identifiers
1 The following 38 regular expressions characterize identifiers that are systematically reserved by some clause this document.

```plaintext
atomic_[a-z][a-zA-Z0-9_]*
ATOMIC_[A-Z][a-zA-Z0-9_]*
__[a-zA-Z_][a-zA-Z0-9_]*
cnd_[a-z][a-zA-Z0-9_]*
DBL_[A-Z][a-zA-Z0-9_]*
DEC128_[A-Z][a-zA-Z0-9_]*
DEC32_[A-Z][a-zA-Z0-9_]*
e[0-9A-Z][a-zA-Z0-9_]*
FE_[A-Z][a-zA-Z0-9_]*
FLT_[A-Z][a-zA-Z0-9_]*
FP_[A-Z][a-zA-Z0-9_]*
INT[a-zA-Z0-9_.]*C
INT[a-zA-Z0-9_.]*MAX
INT[a-zA-Z0-9_.]*MIN
int[a-zA-Z0-9_.]*t
INT[a-zA-Z0-9_.]*WIDTH
is[a-zA-Z0-9_.]*
LC_[A-Z][a-zA-Z0-9_]*
LDBL_[A-Z][a-zA-Z0-9_]*
MATH_[A-Z][a-zA-Z0-9_]*
mem[a-z][a-zA-Z0-9_]*
mtx_[a-z][a-zA-Z0-9_]*
PRI_[a-zA-Z0-9_]*
SCN_[a-zA-Z0-9_]*
SIG_[A-Z][a-zA-Z0-9_]*
SIG[A-Z][a-zA-Z0-9_]*
str[a-zA-Z0-9_]*
thrd_[a-z][a-zA-Z0-9_]*
TIME_[A-Z][a-zA-Z0-9_]*
to[a-zA-Z0-9_]*
tss_[a-zA-Z][a-zA-Z0-9_]*
UINT[a-zA-Z0-9_.]*C
UINT[a-zA-Z0-9_.]*MAX
UINT[a-zA-Z0-9_.]*MAX
uint[a-zA-Z0-9_.]*t
UINT[a-zA-Z0-9_.]*T
wcs[a-zA-Z0-9_]*
```

2 The following 631 identifiers or keywords match these patterns and have particular semantics provided by this document.

```plaintext
__alignas__
__alignas_is_defined
__Alignas__
__alignof_is_defined
__Atomic__
atomic_bool
ATOMIC_BOOL_LOCK_FREE
atomic_char
atomic_char16_t
atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange weak explicit
atomic_exchange
```

508 Portability issues § J.6.1
atomic_exchange_explicit
atomic_fetch_
atomic_fetch('&
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag
atomic_flag_clear
atomic_flag_clear_explicit
ATOMIC_FLAG_INIT
atomic_fetch
atomic_fetch_and_set
atomic_fetch_and_set_explicit
atomic_init
atomic_int
atomic_int C
atomic_int_fast16_t
atomic_int_fast32_t
atomic_int_fast64_t
atomic_int_fast8_t
atomic_int_least16_t
atomic_int_least32_t
atomic_int_least64_t
atomic_int_least8_t
atomic_intmax_t
atomic_intptr_t
atomic_is_lock_free
atomic_uchar
atomic_uint
atomic_uint_fast16_t
atomic_uint_fast32_t
atomic_uint_fast64_t
atomic_uint_fast8_t
atomic_uint_least16_t
atomic_uint_least32_t
atomic_uint_least64_t
atomic_uint_least8_t
atomic_uintmax_t
atomic_uintptr_t
atomic_ullong
atomic_ulong
atomic_ushort
atomic_wchar_t
ATOMIC_VAR_INIT
atomic_wchar_t LOCK_FREE
_Bool
bool true false are defined
_cnd broadcast
_cnd_destroy
_cnd_init
_cnd_signal
_cnd_t
_cnd_timedwait
_cnd_wait
_cComplex
_cComplex I
_cclplusplus
_DATE_
DBL DECIMAL DIG
DBL DIG
DBL EPSILON
DBL HAS SUBNORM
DBL MANT DIG
DBL MAX
DBL MAX 10 EXP
DBL MAX EXP
DBL MIN
DBL MIN 10 EXP
DBL MIN EXP
DBL NORM MAX
DBL TRUE MIN
DEC128 EPSILON
DEC128 MANT DIG
DEC128 MAX
DEC128 MAX EXP
DEC128 MIN
DEC128 MIN EXP
DEC128 MIN EXP
DEC32 EPSILON
DEC32 MANT DIG
DEC32 MAX
DEC32 MAX EXP
DEC32 MIN
DEC32 MIN EXP
DEC32 MIN EXP
DEC64 EPSILON
DEC64 MANT DIG
DEC64 MAX
DEC64 MAX EXP
DEC64 MIN
DEC64 MIN EXP

§ J.6.1 Portability issues
memcpy
memcpy_s
memmove
memmove_s
memory_order
memory_order_acq_rel
memory_order_acquire
memory_order_consume
memory_order_relaxed
memory_order_release
memory_order_seq_cst
memset
memset_s
mtx_destroy
mtx_init
mtx_lock
mtx_plain
mtx_recursive
mtx_t
mtx_timed
mtx_timedlock
mtx_trylock
mtx_unlock
__ndiscard__
_Noreturn
__Pragma
PRI32
PRI64
PRIFAST32
PRIFAST64
PRILEAST32
PRILEAST64
PRIu32
PRIu64
PRIuFAST32
PRIuFAST64
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIuPTR
PRId32
PRId64
PRIdFAST32
PRIdFAST64
PRIdLEAST32
PRIdLEAST64
PRIdMAX
PRIdPTR
PRIi32
PRIi64
PRIiFAST32
PRIiFAST64
PRIiLEAST32
PRIiLEAST64
PRIiMAX
PRIiPTR
PRIo32
PRIo64
PRIoFAST32
PRIoFAST64
PRIoLEAST32
PRIoLEAST64
PRIoMAX
PRIoMIN
PRIopTR
PRIu32
PRIu64
PRIuFAST32
PRIuFAST64
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIuPTR
PRIX32
PRIX64
PRIX_FAST32
PRIX_FAST64
PRIX_LEAST32
PRIX_LEAST64
PRIX_MAX
PRIX_PTR
PRIs32
PRIs64
PRIsFAST32
PRIsFAST64
PRIsLEAST32
PRIsLEAST64
PRIsMAX
PRIsPTR
SCNd32
SCNd64
SCNdFAST32
SCNdFAST64
SCNdLEAST32
SCNdLEAST64
SCNdMAX
SCNdPTR
SCNi32
SCNi64
SCNiFAST32
SCNiFAST64
SCNiLEAST32
SCNiLEAST64
SCNiMAX
SCNiPTR
SCNu32
SCNu64
SCNuFAST32
SCNuFAST64
SCNuLEAST32
SCNuLEAST64
SCNuMAX
SCNuPTR
SIGABRT
SIG_ATOMIC_MAX
SIG_ATOMIC_MIN
SIG_ATOMIC_WIDTH
SIG_ERR
SIG_FPE
SIG_IGN
SIG_INT
SIG_SEGV
SIG_TERM
SIG_DFL
_Static_assert
__STDC__
__STDC_ANALYZABLE__
__STDC_HOSTED__
__STDC_IEC_559__
__STDC_IEC_559_COMPLEX__
__STDC_IEC_60559_BFP__
__STDC_IEC_60559_COMPLEX__
__STDC_IEC_60559_COMPLEXP__
__STDC_IEC_60559_DFP__
__STDC_ISO_10646__
__STDC_LIB_EXT1__
__STDC_MB_MIGHT_NEQ_WC__
__STDC_NO_ATOMICS__
__STDC_NO_COMPLEX__
__STDC_NO_THREADS__
__STDC_NO_VLA__
__STDC_UTF_16__
__STDC_UTF_32__
__STDC_VERSION__
__STDC_VERSION_FENV_H__
__STDC_VERSION_MATH_H__
__STDC_VERSION_STDINT_H__
__STDC_VERSION_STDLIB_H__
__STDC_VERSION_TGMATH_H__
512 Portability issues § J.6.1
__STDC_VERSION_TIME_H__
__STDC_WANT_IEC_60559__
__STDC_WANT_IEC_60559_BFP_EXT__
__STDC_WANT_IEC_60559_DFP_EXT__
__STDC_WANT_LIB_EXT1__

strcat
strcat_s
strchr
strcmp
strcoll
strcpy
strcpy_s
strcspn
strdup
strerror
strerrorlen_s
strerror_s
strerrorrlen_s
tolower
totalorder
totalorder128
totalorderd128
totalorder32
totalorderd32
totalorder64
totalorderd64
totalorderf
totalorderl
totalordermag
totalordermag128
totalordermagd128
totalordermag32
totalordermagd32
totalordermag64
totalordermagd64
totalordermagf
totalordermagl
toupper
towctrans
towlower
towupper
tss_create
tss_delete
tss_dtor_t
tss_get
tss_set
tss_t
UINT16_C
UINT16_MAX
uint16_t
UINT32_C
UINT32_MAX
uint32_t
UINT64_C
UINT64_MAX
uint64_t
UINT8_C
UINT8_MAX
uint8_t
UINT_FAST16_C
uint_fast16_t
UINT_FAST32_C
uint_fast32_t
UINT_FAST64_C
uint_fast64_t
UINT_FAST8_C
uint_fast8_t
UINT_LEAST16_C
uint_least16_t
UINT_LEAST32_C
uint_least32_t
J.6.2 Particular identifiers or keywords

The following identifiers or keywords are not covered by the above and have particular semantics provided by this document.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td></td>
</tr>
<tr>
<td>abort_handler_s</td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td></td>
</tr>
<tr>
<td>acos</td>
<td></td>
</tr>
<tr>
<td>acosd128</td>
<td></td>
</tr>
<tr>
<td>acosd32</td>
<td></td>
</tr>
<tr>
<td>acosd64</td>
<td></td>
</tr>
<tr>
<td>acosf</td>
<td></td>
</tr>
<tr>
<td>acosh</td>
<td></td>
</tr>
<tr>
<td>acoshd128</td>
<td></td>
</tr>
<tr>
<td>acoshd32</td>
<td></td>
</tr>
<tr>
<td>acoshd64</td>
<td></td>
</tr>
<tr>
<td>acoshf</td>
<td></td>
</tr>
<tr>
<td>acos1</td>
<td></td>
</tr>
<tr>
<td>acospi</td>
<td></td>
</tr>
<tr>
<td>acospid128</td>
<td></td>
</tr>
<tr>
<td>acospid32</td>
<td></td>
</tr>
<tr>
<td>acospid64</td>
<td></td>
</tr>
<tr>
<td>acospif</td>
<td></td>
</tr>
<tr>
<td>acospi</td>
<td></td>
</tr>
<tr>
<td>acospid128</td>
<td></td>
</tr>
<tr>
<td>acospid32</td>
<td></td>
</tr>
<tr>
<td>acospid64</td>
<td></td>
</tr>
<tr>
<td>alignas</td>
<td></td>
</tr>
<tr>
<td>aligned_alloc</td>
<td></td>
</tr>
<tr>
<td>alignof</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
</tr>
<tr>
<td>and_eq</td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td></td>
</tr>
<tr>
<td>asind128</td>
<td></td>
</tr>
<tr>
<td>asind32</td>
<td></td>
</tr>
<tr>
<td>asind64</td>
<td></td>
</tr>
<tr>
<td>asinh</td>
<td></td>
</tr>
<tr>
<td>asinhf</td>
<td></td>
</tr>
<tr>
<td>asinhd128</td>
<td></td>
</tr>
<tr>
<td>asinhd32</td>
<td></td>
</tr>
<tr>
<td>asinhd64</td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td></td>
</tr>
<tr>
<td>atan2</td>
<td></td>
</tr>
<tr>
<td>atan2d128</td>
<td></td>
</tr>
<tr>
<td>atan2d32</td>
<td></td>
</tr>
<tr>
<td>atan2d64</td>
<td></td>
</tr>
<tr>
<td>atan2f</td>
<td></td>
</tr>
<tr>
<td>atan2l</td>
<td></td>
</tr>
<tr>
<td>atan2pid128</td>
<td></td>
</tr>
<tr>
<td>atan2pid32</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
</tr>
<tr>
<td>atan2pif</td>
<td></td>
</tr>
<tr>
<td>atan2pif32</td>
<td></td>
</tr>
<tr>
<td>atan2pif64</td>
<td></td>
</tr>
<tr>
<td>atan2pil</td>
<td></td>
</tr>
<tr>
<td>atan2pil32</td>
<td></td>
</tr>
<tr>
<td>atan2pil64</td>
<td></td>
</tr>
<tr>
<td>atanh</td>
<td></td>
</tr>
<tr>
<td>atanhd128</td>
<td></td>
</tr>
<tr>
<td>atanhd32</td>
<td></td>
</tr>
<tr>
<td>atanhd64</td>
<td></td>
</tr>
<tr>
<td>atanhd128</td>
<td></td>
</tr>
<tr>
<td>atanhd32</td>
<td></td>
</tr>
<tr>
<td>atanhd64</td>
<td></td>
</tr>
<tr>
<td>atanhd128</td>
<td></td>
</tr>
<tr>
<td>atanhd32</td>
<td></td>
</tr>
<tr>
<td>atanhd64</td>
<td></td>
</tr>
</tbody>
</table>

514 Portability issues § J.6.2
atanhf  atanhf  clog2pl
atanhl  atanhhl  clogf
atanl  atanhl  clogl
atanpid128  atanh  CMPLX
atanpid32  atanhf  CMPLXF
atanpid64  cbrrt  CMPLXL
atanpif  cbrrtd128  compl
atanpl  cbrrtd32  complex
atexit  cbrrtd64  compoundn
atof  cbrrtf  compoundnd128
atoi  cbrrtl  compoundnd32
atol  compoundn  compoundnd64
atoll  compoundn  compoundnl
at_quick_exit
auto  ccos  const
bitand  ccosf  constraint_handler_t
bitor  ccosf
break  ccosl  continue
bsearch  ccosl
bsearch_s  ccos
btowc  ccosf
BUFFSZ  ccosg
cl6rtomb  ccosg128
c32rtomb  ccosgn
cmplx  cexp
complex  cexpf
compoundn  cexpl
compoundnd128  cexpf
compoundnd32  cexpl
compoundnd64  cexpm1
compoundn  cos
compoundnl  cosd128
char  cosd32
char16_t  cosd64
char32_t  cosf
CHAR_BIT  cosf
CHAR_MAX  copy
CHAR_MIN  copy
CHAR_WIDTH  copy
cimag  copy
cimagf  copy
cimagl  copy
cimagf
cimagl
clearerr  copy
clearerr
clgamma  copyf
clgamma
cllog  copy
cllog
cllogf  cproj
cllog10  cprojl
cllog10p1  cracos
cllog10p1
cllog1p  cracos
cllog2  cracos
Portability issues § J.6.2
exp2
d2
...om2ld128
...om2ld32
...om2ld64
...om2lf
...om2l
...om21
...om21d128
...om21d32
...om21d64
...om21f
...om21l
...om2
...om2d128
...om2d32
...om2d64
...om2f
...om2l
...om2
...om21
...om2d128
...om2d32
...om2d64
...om2f
...om2l
...om2
...om21
Portability issues § J.6.2
long
longjmp
LONG_MAX
LONG_MIN
LONG_WIDTH
lrint
lrintd128
lrintd32
lrintd64
lrintf
lrintl
lround
lroundd128
lroundd32
lroundd64
lroundf
lroundl
L_tmpnam
L_tmpnam_s
main
malloc
math_errhandling
max_align_t
maybe_unused
MB_CUR_MAX
mblen
mbstate_t
mbstatewc
mbstatewcs
mbstatewcs_s
mbtwc
mktime
modf
modfd128
modfd32
modfd64
modff
modfl
mon_decimal_point
mon_grouping
mon_thousands_sep
nan
nand128
nand32
nand64
nanf
nanl
n_cs_precedes
NDEBUG

nearbyint
nearbyintd128
nearbyintd32
nearbyintd64
nearbyintf
nearbyintl
negative_sign
nextafter
nextafterd128
nextafterd32
nextafterd64
nextafterf
nextafterl
nextdown
nextdownd128
nextdownd32
nextdownd64
nextdownf
nextdownl
nexttoward
nexttowardd128
nexttowardd32
nexttowardd64
nexttowardf
nexttowardl
nextup
nextupd128
nextupd32
nextupd64
nextupf
nextupl

p_sep_by_space
p_sign_posn
PTRDIFF_MAX
PTRDIFF_MIN
ptdiff_t

putc
putchar
putwc
qsort
qsort_s
quantize
quantized
quantized128
quantized32
quantized64
quantum
quantum128
quantum32
quantum64
quick_exit
raise
rand
reallocate
register
remainder
remainder128
remainder32
remainder64
remainderf
remainderl
remove
remquo
remquof
remquol
rename
restrict
return
rewind

§ J.6.2 Portability issues
Portability issues § J.6.2
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tm_isdst</td>
<td>ULLONG_MAX</td>
<td>vswscanf</td>
</tr>
<tr>
<td>tm_mday</td>
<td>ULLONG_WIDTH</td>
<td>vswscanf_s</td>
</tr>
<tr>
<td>tm_mday</td>
<td>ULONG_MAX</td>
<td>vfprintf</td>
</tr>
<tr>
<td>tm_mon</td>
<td>ULONG_WIDTH</td>
<td>vfprintf_s</td>
</tr>
<tr>
<td>tmpfile</td>
<td>undef</td>
<td>vsscanf</td>
</tr>
<tr>
<td>tmpfile_s</td>
<td>ungetc</td>
<td>vsscanf_s</td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>ungetwc</td>
<td>WCHAR_MAX</td>
</tr>
<tr>
<td>TMP_MAX_S</td>
<td>union</td>
<td>WCHAR_MIN</td>
</tr>
<tr>
<td>tmnam</td>
<td>unsigned</td>
<td>wchar_t</td>
</tr>
<tr>
<td>tmnam_s</td>
<td>USHR_MAX</td>
<td>WCHAR_WIDTH</td>
</tr>
<tr>
<td>tmsec</td>
<td>USHR_WIDTH</td>
<td>wcrtomb</td>
</tr>
<tr>
<td>tm_wday</td>
<td>va_arg</td>
<td>wcrtomb_s</td>
</tr>
<tr>
<td>tm_yday</td>
<td>va_copy</td>
<td>wcptob</td>
</tr>
<tr>
<td>tm_year</td>
<td>va_end</td>
<td>wcptomb</td>
</tr>
<tr>
<td>true</td>
<td>va_list</td>
<td>wcptomb_s</td>
</tr>
<tr>
<td>trunc</td>
<td>va_start</td>
<td>wcptrans</td>
</tr>
<tr>
<td>truncd128</td>
<td>vfprintff</td>
<td>wcptrans_s</td>
</tr>
<tr>
<td>truncd32</td>
<td>vfprintffs</td>
<td>wctype</td>
</tr>
<tr>
<td>truncd64</td>
<td>vfscanf</td>
<td>wctype_s</td>
</tr>
<tr>
<td>truncf</td>
<td>vfscanf_s</td>
<td>WEOF</td>
</tr>
<tr>
<td>truncl</td>
<td>vfwprintff</td>
<td>while</td>
</tr>
<tr>
<td>TSS_DTOR_ITERATIONS</td>
<td>vfwprintffs</td>
<td>WINT_MAX</td>
</tr>
<tr>
<td>tv_nsec</td>
<td>vfwscanff</td>
<td>WINT_MIN</td>
</tr>
<tr>
<td>tv_sec</td>
<td>vfwscanffs</td>
<td>WINT_T</td>
</tr>
<tr>
<td>typedef</td>
<td>void</td>
<td>WINT_WIDTH</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>volatile</td>
<td>wmemchr</td>
</tr>
<tr>
<td>UCHAR_WIDTH</td>
<td>vprintff</td>
<td>wmemcmp</td>
</tr>
<tr>
<td>ufromfp</td>
<td>vprintffs</td>
<td>wmemcmpy</td>
</tr>
<tr>
<td>ufromfpd128</td>
<td>vscanf</td>
<td>wmemcmpy_s</td>
</tr>
<tr>
<td>ufromfpd32</td>
<td>vscanf_s</td>
<td>wmemmove</td>
</tr>
<tr>
<td>ufromfpd64</td>
<td>vsnprintff</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>ufromfpf</td>
<td>vsnprintffs</td>
<td>wmemset</td>
</tr>
<tr>
<td>ufromfpfl</td>
<td>vsmwprintff</td>
<td>wprintf</td>
</tr>
<tr>
<td>ufromfpx</td>
<td>vsmtpref</td>
<td>wprintf_s</td>
</tr>
<tr>
<td>ufromfpxd128</td>
<td>vsprintf</td>
<td>wscanf</td>
</tr>
<tr>
<td>ufromfpxd32</td>
<td>vsprintfs</td>
<td>wscanf_s</td>
</tr>
<tr>
<td>ufromfpxd64</td>
<td>vsscanf</td>
<td>xor</td>
</tr>
<tr>
<td>ufromfpxf</td>
<td>vswprintff</td>
<td>xor_eq</td>
</tr>
<tr>
<td>ufromfpxl</td>
<td>vswprintfs</td>
<td></td>
</tr>
</tbody>
</table>
Annex K  
(normative)  
Bounds-checking interfaces

K.1 Background

Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.

A common programming style is to declare character arrays large enough to handle most practical cases. However, if these arrays are not large enough to handle the resulting strings, data can be written past the end of the array overwriting other data and program structures. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

Worse, this style of programming has compromised the security of computers and networks. Buffer overflows can often be exploited to run arbitrary code with the permissions of the vulnerable (defective) program.

If the programmer writes runtime checks to verify lengths before calling library functions, then those runtime checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.

This annex provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null terminated.

This annex also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

K.2 Scope

This annex specifies a series of optional extensions that can be useful in the mitigation of security vulnerabilities in programs, and comprise new functions, macros, and types declared or defined in existing standard headers.

An implementation that defines `__STDC_LIB_EXT1__` shall conform to the specifications in this annex.

Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of Clause 7.

K.3 Library

K.3.1 Introduction

K.3.1.1 Standard headers

The functions, macros, and types declared or defined in K.3 and its subclauses are not declared or defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant 0 at the point in the source file where the appropriate header is first included.

The functions, macros, and types declared or defined in K.3 and its subclauses are declared and defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant 1 at the point in the source file where the appropriate header is first included.

Implementations that do not define `__STDC_LIB_EXT1__` are not required to conform to these specifications.
It is implementation-defined whether the functions, macros, and types declared or defined in K.3 and its subclauses are declared or defined by their respective headers if __STDC_WANT_LIB_EXT1__ is not defined as a macro at the point in the source file where the appropriate header is first included.

Within a preprocessing translation unit, __STDC_WANT_LIB_EXT1__ shall be defined identically for all inclusions of any headers from Subclause K.3. If __STDC_WANT_LIB_EXT1__ is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive were used.

K.3.1.2 Reserved identifiers

Each macro name in any of the following subclauses is reserved for use as specified if it is defined by any of its associated headers when included; unless explicitly stated otherwise (see 7.1.4).

All identifiers with external linkage in any of the following subclauses are reserved for use as identifiers with external linkage if any of them are used by the program. None of them are reserved if none of them are used.

Each identifier with file scope listed in any of the following subclauses is reserved for use as a macro name and as an identifier with file scope in the same name space if it is defined by any of its associated headers when included.

K.3.1.3 Use of errno

An implementation may set errno for the functions defined in this annex, but is not required to.

K.3.1.4 Runtime-constraint violations

Most functions in this annex include as part of their specification a list of runtime-constraints. These runtime-constraints are requirements on the program using the library.

Implementations shall verify that the runtime-constraints for a function are not violated by the program. If a runtime-constraint is violated, the implementation shall call the currently registered runtime-constraint handler (see set_constraint_handler_s in <stdlib.h>). Multiple runtime-constraint violations in the same call to a library function result in only one call to the runtime-constraint handler. It is unspecified which one of the multiple runtime-constraint violations cause the handler to be called.

If the runtime-constraints section for a function states an action to be performed when a runtime-constraint violation occurs, the function shall perform the action before calling the runtime-constraint handler. If the runtime-constraints section lists actions that are prohibited when a runtime-constraint violation occurs, then such actions are prohibited to the function both before calling the handler and after the handler returns.

The runtime-constraint handler might not return. If the handler does return, the library function whose runtime-constraint was violated shall return some indication of failure as given by the returns section in the function’s specification.

K.3.2 Errors <errno.h>

The header <errno.h> defines a type.

The type is

```c
errno_t
```

Future revisions of this document might define meanings for other values of __STDC_WANT_LIB_EXT1__.

Subclause 7.1.3 reserves certain names and patterns of names that an implementation can use in headers. All other names are not reserved, and a conforming implementation is not permitted to use them. While some of the names defined in K.3 and its subclauses are reserved, others are not. If an unreserved name is defined in a header when __STDC_WANT_LIB_EXT1__ is defined as 0, the implementation is not conforming.

Although runtime-constraints replace many cases of undefined behavior, undefined behavior still exists in this annex. Implementations are free to detect any case of undefined behavior and treat it as a runtime-constraint violation by calling the runtime-constraint handler. This license comes directly from the definition of undefined behavior.
which is type `int`.\(^{411}\)

**K.3.3 Common definitions `<stddef.h>`**

The header `<stddef.h>` defines a type.

The type is

```c
rsize_t
```

which is the type `size_t`.\(^{412}\)

**K.3.4 Integer types `<stdint.h>`**

The header `<stdint.h>` defines a macro.

The macro is

```c
RSIZE_MAX
```

which expands to a value\(^{413}\) of type `size_t`. Functions that have parameters of type `rsize_t` consider it a runtime-constraint violation if the values of those parameters are greater than `RSIZE_MAX`.

**Recommended practice**

Extremely large object sizes are frequently a sign that an object’s size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`.

For those reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.

**K.3.5 Input/output `<stdio.h>`**

The header `<stdio.h>` defines several macros and two types.

The macros are

```c
L_tmpnam_s
```

which expands to an integer constant expression that is the size needed for an array of `char` large enough to hold a temporary file name string generated by the `tmpnam_s` function;

```c
TMP_MAX_S
```

which expands to an integer constant expression that is the maximum number of unique file names that can be generated by the `tmpnam_s` function.

The types are

```c
errno_t
```

which is type `int`; and

\(^{411}\)As a matter of programming style, `errno_t` can be used as the type of something that deals only with the values that might be found in `errno`. For example, a function which returns the value of `errno` could be declared as having the return type `errno_t`.

\(^{412}\)See the description of the `RSIZE_MAX` macro in `<stdint.h>`.

\(^{413}\)The macro `RSIZE_MAX` need not expand to a constant expression.
K.3.5.1 Operations on files
K.3.5.1.1 The tmpfile_s function
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpfile_s(FILE * restrict * restrict streamptr);
```

Runtime-constraints

1. `streamptr` shall not be a null pointer.
2. If there is a runtime-constraint violation, `tmpfile_s` does not attempt to create a file.

Description

3. The `tmpfile_s` function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "wb+" mode with the meaning that mode has in the `fopen_s` function (including the mode's effect on exclusive access and file permissions).

4. If the file was created successfully, then the pointer to `FILE` pointed to by `streamptr` will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to `FILE` pointed to by `streamptr` will be set to a null pointer.

Recommended practice

It should be possible to open at least `TMP_MAX_S` temporary files during the lifetime of the program (this limit may be shared with `tmpnam_s`) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (`FOPEN_MAX`).

Returns

5. The `tmpfile_s` function returns zero if it created the file. If it did not create the file or there was a runtime-constraint violation, `tmpfile_s` returns a nonzero value.

K.3.5.1.2 The tmpnam_s function
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t tmpnam_s(char *s, rsize_t maxsize);
```

Runtime-constraints

1. `s` shall not be a null pointer. `maxsize` shall be less than or equal to `RSIZE_MAX`. `maxsize` shall be greater than the length of the generated file name string.

Description

3. The `tmpnam_s` function generates a string that is a valid file name and that is not the same as the name of an existing file. The function is potentially capable of generating `TMP_MAX_S` different strings, but any or all of them may already be in use by existing files and thus not be suitable return values. The lengths of these strings shall be less than the value of the `L_tmpnam_s` macro.

4. The `tmpnam_s` function generates a different string each time it is called.

Files created using strings generated by the `tmpnam_s` function are temporary only in the sense that their names are not expected to collide with those generated by conventional naming rules for the implementation. It is still necessary to use the `remove` function to remove such files when their use is ended, and before program termination.

§ K.3.5.1.2 Bounds-checking interfaces 525
It is assumed that \texttt{s} points to an array of at least \texttt{maxsize} characters. This array will be set to generated string, as specified below.

The implementation shall behave as if no library function except \texttt{tmpnam} calls the \texttt{tmpnam_s} function.\(^1\)

**Recommended practice**

After a program obtains a file name using the \texttt{tmpnam_s} function and before the program creates a file with that name, the possibility exists that someone else may create a file with that same name. To avoid this race condition, the \texttt{tmpfile_s} function should be used instead of \texttt{tmpnam_s} when possible. One situation that requires the use of the \texttt{tmpnam_s} function is when the program needs to create a temporary directory rather than a temporary file.

Implementations should take care in choosing the patterns used for names returned by \texttt{tmpnam_s}. For example, making a thread ID part of the names avoids the race condition and possible conflict when multiple programs run simultaneously by the same user generate the same temporary file names.

**Returns**

If no suitable string can be generated, or if there is a runtime-constraint violation, the \texttt{tmpnam_s} function:

- if \texttt{s} is not null and \texttt{maxsize} is both greater than zero and not greater than \texttt{RSIZE_MAX}, writes a null character to \texttt{s[0]}
- returns a nonzero value.

Otherwise, the \texttt{tmpnam_s} function writes the string in the array pointed to by \texttt{s} and returns zero.

**Environmental limits**

The value of the macro \texttt{TMP_MAX_S} shall be at least 25.

**K.3.5.2 File access functions**

**K.3.5.2.1 The \texttt{fopen_s} function**

**Synopsis**

\begin{verbatim}
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
errno_t fopen_s(FILE * restrict * restrict streamptr,
                const char * restrict filename, const char * restrict mode);
\end{verbatim}

**Runtime-constraints**

None of \texttt{streamptr}, \texttt{filename}, or \texttt{mode} shall be a null pointer.

If there is a runtime-constraint violation, \texttt{fopen_s} does not attempt to open a file. Furthermore, if \texttt{streamptr} is not a null pointer, \texttt{fopen_s} sets \texttt{*streamptr} to the null pointer.

**Description**

The \texttt{fopen_s} function opens the file whose name is the string pointed to by \texttt{filename}, and associates a stream with it.

The \texttt{mode} string shall be as described for \texttt{fopen}, with the addition that modes starting with the character ‘\texttt{w}’ or ‘\texttt{a}’ may be preceded by the character ‘\texttt{u}’, see below:

- \texttt{uw}: truncate to zero length or create text file for writing, default permissions
- \texttt{uwx}: create text file for writing, default permissions
- \texttt{ua}: append; open or create text file for writing at end-of-file, default permissions

---

\(^1\)An implementation can have \texttt{tmpnam} call \texttt{tmpnam_s} (perhaps so there is only one naming convention for temporary files), but this is not required.
uwb  truncate to zero length or create binary file for writing, default permissions
uwbx create binary file for writing, default permissions
uab append; open or create binary file for writing at end-of-file, default permissions
uw+ truncate to zero length or create text file for update, default permissions
uw+x create text file for update, default permissions
ua+ append; open or create text file for update, writing at end-of-file, default permissions
uw+b or uwb+ truncate to zero length or create binary file for update, default permissions
uw+bx or uwb+ create binary file for update, default permissions
ua+b or uab+ append; open or create binary file for update, writing at end-of-file, default permissions

6 Opening a file with exclusive mode (‘x’ as the last character in the mode argument) fails if the file already exists or cannot be created.

7 To the extent that the underlying system supports the concepts, files opened for writing shall be opened with exclusive (also known as non-shared) access. If the file is being created, and the first character of the mode string is not ‘u’, to the extent that the underlying system supports it, the file shall have a file permission that prevents other users on the system from accessing the file. If the file is being created and first character of the mode string is ‘u’, then by the time the file has been closed, it shall have the system default file access permissions.416)

8 If the file was opened successfully, then the pointer to FILE pointed to by streamptr will be set to the pointer to the object controlling the opened file. Otherwise, the pointer to FILE pointed to by streamptr will be set to a null pointer.

Returns
9 The fopen_s function returns zero if it opened the file. If it did not open the file or if there was a runtime-constraint violation, fopen_s returns a nonzero value.

K.3.5.2.2 The freopen_s function

Synopsis

```c
#include <stdio.h>
errno_t freopen_s(FILE * restrict * restrict newstreamptr,
                   const char * restrict filename, const char * restrict mode,
                   FILE * restrict stream);
```

Runtime-constraints
2 None of newstreamptr, mode, and stream shall be a null pointer.
3 If there is a runtime-constraint violation, freopen_s neither attempts to close any file associated with stream nor attempts to open a file. Furthermore, if newstreamptr is not a null pointer, fopen_s sets *newstreamptr to the null pointer.

Description
4 The freopen_s function opens the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument has the same meaning as in the fopen_s function (including the mode’s effect on exclusive access and file permissions).

§ K.3.5.2.2 Bounds-checking interfaces 527

416) These are the same permissions that the file would have been created with by fopen.
If `filename` is a null pointer, the `freopen_s` function attempts to change the mode of the stream to that specified by `mode`, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

The `freopen_s` function first attempts to close any file that is associated with `stream`. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

If the file was opened successfully, then the pointer to `FILE` pointed to by `newstreamptr` will be set to the value of `stream`. Otherwise, the pointer to `FILE` pointed to by `newstreamptr` will be set to a null pointer.

**Returns**

The `freopen_s` function returns zero if it opened the file. If it did not open the file or there was a runtime-constraint violation, `freopen_s` returns a nonzero value.

### K.3.5.3 Formatted input/output functions

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

#### K.3.5.3.1 The `fprintf_s` function

**Synopsis**

```c
#include <stdio.h>

int fprintf_s(FILE * restrict stream, const char * restrict format, ...);
```

**Runtime-constraints**

Neither `stream` nor `format` shall be a null pointer. The `%n` specifier (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to `fprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

If there is a runtime-constraint violation, the `fprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `fprintf_s` produced output before discovering the runtime-constraint violation.

**Description**

The `fprintf_s` function is equivalent to the `fprintf` function except for the explicit runtime-constraints listed above.

**Returns**

The `fprintf_s` function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

#### K.3.5.3.2 The `fscanf_s` function

**Synopsis**

```c
#include <stdio.h>

int fscanf_s(FILE * restrict stream, const char * restrict format, ...);
```

**Runtime-constraints**

Neither `stream` nor `format` shall be a null pointer. Any argument indirectioned though in order to store converted input shall not be a null pointer.

If there is a runtime-constraint violation, the `fscanf_s` function does not attempt to perform

---

417) It is not a runtime-constraint violation for the characters `%n` to appear in sequence in the string pointed at by `format` when those characters are not interpreted as a `%n` specifier. For example, if the entire format string was `%%n`.

418) Because an implementation can treat any undefined behavior as a runtime-constraint violation, an implementation can treat any unsupported specifiers in the string pointed to by `format` as a runtime-constraint violation.

419) Because an implementation can treat any undefined behavior as a runtime-constraint violation, an implementation can
further input, and it is unspecified to what extent fscanf_s performed input before discovering the runtime-constraint violation.

**Description**

The fscanf_s function is equivalent to fscanf except that the c, s, and [ conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a *). The first of these arguments is the same as for fscanf. That argument is immediately followed in the argument list by the second argument, which has type rsize_t and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.  

A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

**Returns**

The fscanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the fscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

**EXAMPLE 1** The call:

```c
#include <stdio.h>
int n, i; float x; char name[50];
int n = fscanf_s(stdin, "%d%f%s", &i, &x, name, (rsize_t) 50);
```

with the input line:

25 54.32E-1 thompson

will assign to n the value 3, to i the value 25, to x the value 5.432, and to name the sequence thompson\0.

**EXAMPLE 2** The call:

```c
#include <stdio.h>
int n; char s[5];
int n = fscanf_s(stdin, "%s", s, sizeof s);
```

with the input line:

hello

will assign to n the value 0 since a matching failure occurred because the sequence hello\0 requires an array of six characters to store it.

**K.3.5.3.3 The printf_s function**

**Synopsis**

```c
#include <stdio.h>
int printf_s(const char * restrict format, ...);
```

treat any unsupported specifiers in the string pointed to by format as a runtime-constraint violation.  

If the format is known at translation time, an implementation can issue a diagnostic for any argument used to store the result from a c, s, or [ conversion specifier if that argument is not followed by an argument of a type compatible with rsize_t. A limited amount of checking can be done if even if the format is not known at translation time. For example, an implementation could issue a diagnostic for each argument after format that has of type pointer to one of char, signed char, unsigned char, or void that is not followed by an argument of a type compatible with rsize_t. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the #h length modifier, a length argument is expected to follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following format that did not have a type compatible with rsize_t.
Runtime-constraints

2 format shall not be a null pointer. The %n specifier\(^{421}\) (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to printf_s corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the printf_s function does not attempt to produce further output, and it is unspecified to what extent printf_s produced output before discovering the runtime-constraint violation.

Description

4 The printf_s function is equivalent to the printf function except for the explicit runtime-constraints listed above.

Returns

5 The printf_s function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.4 The scanf_s function

Synopsis

1

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int scanf_s(const char * restrict format, ...);
```

Runtime-constraints

2 format shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the scanf_s function does not attempt to perform further input, and it is unspecified to what extent scanf_s performed input before discovering the runtime-constraint violation.

Description

4 The scanf_s function is equivalent to fscanf_s with the argument stdin interposed before the arguments to scanf_s.

Returns

5 The scanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the scanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.5 The snprintf_s function

Synopsis

1

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int snprintf_s(char * restrict s, rsize_t n, const char * restrict format, ...);
```

Runtime-constraints

2 Neither s nor format shall be a null pointer. n shall neither equal zero nor be greater than RSIZE_MAX. The %n specifier\(^{422}\) (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to snprintf_s corresponding to a %s specifier shall not be a null pointer. No encoding error shall occur.

---

\(^{421}\)It is not a runtime-constraint violation for the characters %n to appear in sequence in the string pointed at by format when those characters are not interpreted as a %n specifier. For example, if the entire format string was %n.

\(^{422}\)It is not a runtime-constraint violation for the characters %n to appear in sequence in the string pointed at by format when those characters are not interpreted as a %n specifier. For example, if the entire format string was %n.
3 If there is a runtime-constraint violation, then if \( s \) is not a null pointer and \( n \) is greater than zero and not greater than \( \texttt{RSIZE\_MAX} \), then the \texttt{snprintf\_s} function sets \( s[0] \) to the null character.

Description
4 The \texttt{snprintf\_s} function is equivalent to the \texttt{snprintf} function except for the explicit runtime-constraints listed above.

5 The \texttt{snprintf\_s} function, unlike \texttt{sprintf\_s}, will truncate the result to fit within the array pointed to by \( s \).

Returns
6 The \texttt{snprintf\_s} function returns the number of characters that would have been written had \( n \) been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than \( n \).

K.3.5.3.6 The \texttt{sprintf\_s} function
Synopsis
1

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sprintf_s(char * restrict s, size_t n, const char * restrict format, ...);
```

Runtime-constraints
2 Neither \( s \) nor \texttt{format} shall be a null pointer. \( n \) shall neither equal zero nor be greater than \( \texttt{RSIZE\_MAX} \). The number of characters (including the trailing null) required for the result to be written to the array pointed to by \( s \) shall not be greater than \( n \). The \%n specifier\(^{423}\) (modified or not by flags, field width, or precision) shall not appear in the string pointed to by \texttt{format}. Any argument to \texttt{sprintf\_s} corresponding to a \%s specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if \( s \) is not a null pointer and \( n \) is greater than zero and not greater than \( \texttt{RSIZE\_MAX} \), then the \texttt{sprintf\_s} function sets \( s[0] \) to the null character.

Description
4 The \texttt{sprintf\_s} function is equivalent to the \texttt{sprintf} function except for the parameter \( n \) and the explicit runtime-constraints listed above.

5 The \texttt{sprintf\_s} function, unlike \texttt{snprintf\_s}, treats a result too big for the array pointed to by \( s \) as a runtime-constraint violation.

Returns
6 If no runtime-constraint violation occurred, the \texttt{sprintf\_s} function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, \texttt{sprintf\_s} returns a negative value. If any other runtime-constraint violation occurred, \texttt{sprintf\_s} returns zero.

\(^{423}\)It is not a runtime-constraint violation for the characters \%n to appear in sequence in the string pointed at by \texttt{format} when those characters are not a interpreted as a \%n specifier. For example, if the entire format string was \%n.
K.3.5.3.7 The sscanf_s function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
int sscanf_s(const char * restrict s, const char * restrict format, ...);
```

Runtime-constraints

1. Neither `s` nor `format` shall be a null pointer. Any argument indirection through in order to store converted input shall not be a null pointer.

2. If there is a runtime-constraint violation, the `sscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `sscanf_s` performed input before discovering the runtime-constraint violation.

Description

3. The `sscanf_s` function is equivalent to `fscanf_s`, except that input is obtained from a string (specified by the argument `s`) rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the `fscanf_s` function. If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

4. The `sscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `sscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.8 The vfprintf_s function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfprintf_s(FILE * restrict stream, const char * restrict format, va_list arg);
```

Runtime-constraints

1. Neither `stream` nor `format` shall be a null pointer. The `%n` specifier (modified or not by flags, field width, or precision) shall not appear in the string pointed to by `format`. Any argument to `vfprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

2. If there is a runtime-constraint violation, the `vfprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `vfprintf_s` produced output before discovering the runtime-constraint violation.

Description

3. The `vfprintf_s` function is equivalent to the `vfprintf` function except for the explicit runtime-constraints listed above.

Returns

4. The `vfprintf_s` function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.
K.3.5.3.9  The vfscanf_s function

Synopsis
1

#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vfscanf_s(FILE *restrict stream, const char *restrict format, va_list arg);

Runtime-constraints
2 Neither stream nor format shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.
3 If there is a runtime-constraint violation, the vfscanf_s function does not attempt to perform further input, and it is unspecified to what extent vfscanf_s performed input before discovering the runtime-constraint violation.

Description
4 The vfscanf_s function is equivalent to fscanf_s, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vfscanf_s function does not invoke the va_end macro.425

Returns
5 The vfscanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the vfscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.10  The vprintf_s function

Synopsis
1

#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vprintf_s(const char *restrict format, va_list arg);

Runtime-constraints
2 format shall not be a null pointer. The %n specifier (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to vprintf_s corresponding to a % specifier shall not be a null pointer.
3 If there is a runtime-constraint violation, the vprintf_s function does not attempt to produce further output, and it is unspecified to what extent vprintf_s produced output before discovering the runtime-constraint violation.

Description
4 The vprintf_s function is equivalent to the vprintf function except for the explicit runtime-constraints listed above.

Returns
5 The vprintf_s function returns the number of characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.5.3.11  The vscanf_s function

425)As the functions vfprintf_s, vfscanf_s, vprintf_s, vscanf_s, vsnprintf_s, vsprintf_s, and vsscanf_s invoke the va_arg macro, the value of arg after the return is indeterminate.
426)It is not a runtime-constraint violation for the characters %n to appear in sequence in the string pointed at by format when those characters are not interpreted as a %n specifier. For example, if the entire format string was %\n.

§ K.3.5.3.11  Bounds-checking interfaces 533
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vscanf_s(const char * restrict format, va_list arg);
```

Runtime-constraints

2 format shall not be a null pointer. Any argument indirection though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the vscanf_s function does not attempt to perform further input, and it is unspecified to what extent vscanf_s performed input before discovering the runtime-constraint violation.

Description

4 The vscanf_s function is equivalent to scanf_s, with the variable argument list replaced by arg, which shall have been initialized by the va_start macro (and possibly subsequent va_arg calls). The vscanf_s function does not invoke the va_end macro.427

Returns

5 The vscanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the vscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.3.12 The vsnprintf_s function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsnprintf_s(char * restrict s, rsize_t n, const char * restrict format, va_list arg);
```

Runtime-constraints

2 Neither s nor format shall be a null pointer. n shall neither equal zero nor be greater than RSIZE_MAX. The %n specifier (modified or not by flags, field width, or precision) shall not appear in the string pointed to by format. Any argument to vsnprintf_s corresponding to a %s specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if s is not a null pointer and n is greater than zero and not greater than RSIZE_MAX, then the vsnprintf_s function sets s[0] to the null character.

Description

4 The vsnprintf_s function is equivalent to the vsnprintf function except for the explicit runtime-constraints listed above.

5 The vsnprintf_s function, unlike vsprintf_s, will truncate the result to fit within the array pointed to by s.

Returns

6 The vsnprintf_s function returns the number of characters that would have been written had n been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and
only if the returned value is nonnegative and less than \( n \).

K.3.5.3.13 The \texttt{vssprintf} function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vssprintf_s(char * restrict s, rsize_t n, const char * restrict format, va_list arg);
```

Runtime-constraints

2 Neither \texttt{s} nor \texttt{format} shall be a null pointer. \texttt{n} shall neither equal zero nor be greater than \texttt{RSIZE_MAX}. The number of characters (including the trailing null) required for the result to be written to the array pointed to by \texttt{s} shall not be greater than \texttt{n}. The \%n specifier\(^{429}\) (modified or not by flags, field width, or precision) shall not appear in the string pointed to by \texttt{format}. Any argument to \texttt{vssprintf} corresponding to a \%s specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if \texttt{s} is not a null pointer and \texttt{n} is greater than zero and not greater than \texttt{RSIZE_MAX}, then the \texttt{vssprintf} function sets \texttt{s[0]} to the null character.

Description

4 The \texttt{vssprintf} function is equivalent to the \texttt{vssprintf} function except for the parameter \texttt{n} and the explicit runtime-constraints listed above.

5 The \texttt{vssprintf} function, unlike \texttt{vssprintf}, treats a result too big for the array pointed to by \texttt{s} as a runtime-constraint violation.

Returns

6 If no runtime-constraint violation occurred, the \texttt{vssprintf} function returns the number of characters written in the array, not counting the terminating null character. If an encoding error occurred, \texttt{vssprintf} returns a negative value. If any other runtime-constraint violation occurred, \texttt{vssprintf} returns zero.

K.3.5.3.14 The \texttt{vsscanf} function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
int vsscanf_s(const char * restrict s, const char * restrict format, va_list arg);
```

Runtime-constraints

2 Neither \texttt{s} nor \texttt{format} shall be a null pointer. Any argument indirection though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the \texttt{vsscanf} function does not attempt to perform further input, and it is unspecified to what extent \texttt{vsscanf} performed input before discovering the runtime-constraint violation.

Description

4 The \texttt{vsscanf} function is equivalent to \texttt{sscanf}, with the variable argument list replaced by \texttt{arg}, which shall have been initialized by the \texttt{va_start} macro (and possibly subsequent \texttt{va_arg} calls). The \texttt{vsscanf} function does not invoke the \texttt{va_end} macro.\(^{430}\)

\(^{429}\) It is not a runtime-constraint violation for the characters \%n to appear in sequence in the string pointed at by \texttt{format} when those characters are not a interpreted as a \%n specifier. For example, if the entire format string was \%n.

\(^{430}\) As the functions \texttt{vfprintf}, \texttt{vfscanf}, \texttt{vprintf}, \texttt{vscanf}, \texttt{vsnprintf}, \texttt{vsprintf}, and \texttt{vsscanf} invoke the \texttt{va_arg} macro, the value of \texttt{arg} after the return is indeterminate.

\section*{§ K.3.5.3.14} Bounds-checking interfaces
Returns
5 The \texttt{vsscanf\_s} function returns the value of the macro \texttt{EOF} if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the \texttt{vscanf\_s} function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.5.4 Character input/output functions
K.3.5.4.1 The \texttt{gets\_s} function

Synopsis
1

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
char *gets_s(char *s, rsize_t n);
```

Runtime-constraints
2 \texttt{s} shall not be a null pointer. \texttt{n} shall neither be equal to zero nor be greater than \texttt{RSIZE\_MAX}. A new-line character, end-of-file, or read error shall occur within reading \texttt{n-1} characters from \texttt{stdin}. \(^{431)}
3 If there is a runtime-constraint violation, characters are read and discarded from \texttt{stdin} until a new-line character is read, or end-of-file or a read error occurs, and if \texttt{s} is not a null pointer, \texttt{s[0]} is set to the null character.

Description
4 The \texttt{gets\_s} function reads at most one less than the number of characters specified by \texttt{n} from the stream pointed to by \texttt{stdin}, into the array pointed to by \texttt{s}. No additional characters are read after a new-line character (which is discarded) or after end-of-file. The discarded new-line character does not count towards number of characters read. A null character is written immediately after the last character read into the array.
5 If end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then \texttt{s[0]} is set to the null character, and the other elements of \texttt{s} take unspecified values.

Recommended practice
6 The \texttt{fgets} function allows properly-written programs to safely process input lines too long to store in the result array. In general this requires that callers of \texttt{fgets} pay attention to the presence or absence of a new-line character in the result array. Consider using \texttt{fgets} (along with any needed processing based on new-line characters) instead of \texttt{gets\_s}.

Returns
7 The \texttt{gets\_s} function returns \texttt{s} if successful. If there was a runtime-constraint violation, or if end-of-file is encountered and no characters have been read into the array, or if a read error occurs during the operation, then a null pointer is returned.

\(^{431)}\)The \texttt{gets\_s} function, unlike the historical \texttt{gets} function, makes it a runtime-constraint violation for a line of input to overflow the buffer to store it. Unlike the \texttt{fgets} function, \texttt{gets\_s} maintains a one-to-one relationship between input lines and successful calls to \texttt{gets\_s}. Programs that use \texttt{gets} Expect such a relationship.

\section*{Bounds-checking interfaces}
K.3.6 General utilities `<stdlib.h>`

The header `<stdlib.h>` defines three types.

The types are

```c
errno_t
```

which is type `int`; and

```c
rsize_t
```

which is the type `size_t`; and

```c
constraint_handler_t
```

which has the following definition

```c
typedef void (*constraint_handler_t)(
    const char * restrict msg,
    void * restrict ptr,
    errno_t error);
```

K.3.6.1 Runtime-constraint handling

K.3.6.1.1 The `set_constraint_handler_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
```

Description

The `set_constraint_handler_s` function sets the runtime-constraint handler to be `handler`. The runtime-constraint handler is the function to be called when a library function detects a runtime-constraint violation. Only the most recent handler registered with `set_constraint_handler_s` is called when a runtime-constraint violation occurs.

When the handler is called, it is passed the following arguments in the following order:

1. A pointer to a character string describing the runtime-constraint violation.
2. A null pointer or a pointer to an implementation defined object.
3. If the function calling the handler has a return type declared as `errno_t`, the return value of the function is passed. Otherwise, a positive value of type `errno_t` is passed.

The implementation has a default constraint handler that is used if no calls to the `set_constraint_handler_s` function have been made. The behavior of the default handler is implementation-defined, and it may cause the program to exit or abort.

If the `handler` argument to `set_constraint_handler_s` is a null pointer, the implementation default handler becomes the current constraint handler.

Returns

The `set_constraint_handler_s` function returns a pointer to the previously registered handler.432)

---

432) If the previous handler was registered by calling `set_constraint_handler_s` with a null pointer argument, a pointer to the implementation default handler is returned (not NULL).
K.3.6.1.2 The `abort_handler_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void abort_handler_s(const char * restrict msg, void * restrict ptr,
                     errno_t error);
```

Description

A pointer to the `abort_handler_s` function shall be a suitable argument to the `set_constraint_handler_s` function.

The `abort_handler_s` function writes a message on the standard error stream in an implementation-defined format. The message shall include the string pointed to by `msg`. The `abort_handler_s` function then calls the `abort` function.\(^\text{433}\)

Returns

The `abort_handler_s` function does not return to its caller.

K.3.6.1.3 The `ignore_handler_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void ignore_handler_s(const char * restrict msg, void * restrict ptr,
                      errno_t error);
```

Description

A pointer to the `ignore_handler_s` function shall be a suitable argument to the `set_constraint_handler_s` function.

The `ignore_handler_s` function simply returns to its caller.\(^\text{434}\)

Returns

The `ignore_handler_s` function returns no value.

K.3.6.2 Communication with the environment

K.3.6.2.1 The `getenv_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t getenv_s(size_t * restrict len, char * restrict value, rsize_t maxsize,
                 const char * restrict name);
```

Runtime-constraints

- `name` shall not be a null pointer. `maxsize` shall not be greater than `RSIZE_MAX`. If `maxsize` is not equal to zero, then `value` shall not be a null pointer.

\(^\text{433}\)Many implementations invoke a debugger when the `abort` function is called.

\(^\text{434}\)If the runtime-constraint handler is set to the `ignore_handler_s` function, any library function in which a runtime-constraint violation occurs will return to its caller. The caller can determine whether a runtime-constraint violation occurred based on the library function’s specification (usually, the library function returns a nonzero `errno_t`).
If there is a runtime-constraint violation, the integer pointed to by \texttt{len} is set to 0 (if \texttt{len} is not null), and the environment list is not searched.

**Description**

The \texttt{getenv\_s} function searches an environment list, provided by the host environment, for a string that matches the string pointed to by \texttt{name}.

If that name is found then \texttt{getenv\_s} performs the following actions. If \texttt{len} is not a null pointer, the length of the string associated with the matched list member is stored in the integer pointed to by \texttt{len}. If the length of the associated string is less than \texttt{maxsize}, then the associated string is copied to the array pointed to by \texttt{value}.

If that name is not found then \texttt{getenv\_s} performs the following actions. If \texttt{len} is not a null pointer, zero is stored in the integer pointed to by \texttt{len}. If \texttt{maxsize} is greater than zero, then \texttt{value[0]} is set to the null character.

The set of environment names and the method for altering the environment list are implementation-defined. The \texttt{getenv\_s} function need not avoid data races with other threads of execution that modify the environment list.\(^{435}\)

**Returns**

The \texttt{getenv\_s} function returns zero if the specified \texttt{name} is found and the associated string was successfully stored in \texttt{value}. Otherwise, a nonzero value is returned.

### K.3.6.3 Searching and sorting utilities

These utilities make use of a comparison function to search or sort arrays of unspecified type. Where an argument declared as \texttt{size\_t nmemb} specifies the length of the array for a function, if \texttt{nmemb} has the value zero on a call to that function, then the comparison function is not called, a search finds no matching element, sorting performs no rearrangement, and the pointer to the array may be null.

The implementation shall ensure that the second argument of the comparison function (when called from \texttt{bsearch\_s}), or both arguments (when called from \texttt{qsort\_s}), are pointers to elements of the array.\(^{436}\) The first argument when called from \texttt{bsearch\_s} shall equal \texttt{key}.

The comparison function shall not alter the contents of either the array or search key. The implementation may reorder elements of the array between calls to the comparison function, but shall not otherwise alter the contents of any individual element.

When the same objects (consisting of \texttt{size} bytes, irrespective of their current positions in the array) are passed more than once to the comparison function, the results shall be consistent with one another. That is, for \texttt{qsort\_s} they shall define a total ordering on the array, and for \texttt{bsearch\_s} the same object shall always compare the same way with the key.

A sequence point occurs immediately before and immediately after each call to the comparison function, and also between any call to the comparison function and any movement of the objects passed as arguments to that call.

#### K.3.6.3.1 The \texttt{bsearch\_s} function

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
void *bsearch_s(const void *key, const void *base, rsize_t nmemb, rsize_t size, int (*compar)(const void *k, const void *y, void *context), void *context);
```

\(^{435}\)Many implementations provide non-standard functions that modify the environment list.

\(^{436}\)That is, if the value passed is \texttt{p}, then the following expressions are always valid and nonzero:

```c
((char *)p - (char *)base) % size == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nmemb * size
```
Runtime-constraints
2 Neither `nmemb` nor `size` shall be greater than `RSIZE_MAX`. If `nmemb` is not equal to zero, then none of `key`, `base`, or `compar` shall be a null pointer.
3 If there is a runtime-constraint violation, the `bsearch_s` function does not search the array.

Description
4 The `bsearch_s` function searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element that matches the object pointed to by `key`. The size of each element of the array is specified by `size`.
5 The comparison function pointed to by `compar` is called with three arguments. The first two point to the `key` object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the `key` object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.\(^{437}\) The third argument to the comparison function is the `context` argument passed to `bsearch_s`. The sole use of `context` by `bsearch_s` is to pass it to the comparison function.\(^{438}\)

Returns
6 The `bsearch_s` function returns a pointer to a matching element of the array, or a null pointer if no match is found or there is a runtime-constraint violation. If two elements compare as equal, which element is matched is unspecified.

K.3.6.3.2 The `qsort_s` function

Synopsis
1
```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t qsort_s(void *base, rsize_t nmemb, rsize_t size,
    int (*compar)(const void *x, const void *y, void *context),
    void *context);
```

Runtime-constraints
2 Neither `nmemb` nor `size` shall be greater than `RSIZE_MAX`. If `nmemb` is not equal to zero, then neither `base` nor `compar` shall be a null pointer.
3 If there is a runtime-constraint violation, the `qsort_s` function does not sort the array.

Description
4 The `qsort_s` function sorts an array of `nmemb` objects, the initial element of which is pointed to by `base`. The size of each object is specified by `size`.
5 The contents of the array are sorted into ascending order according to a comparison function pointed to by `compar`, which is called with three arguments. The first two point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. The third argument to the comparison function is the `context` argument passed to `qsort_s`. The sole use of `context` by `qsort_s` is to pass it to the comparison function.\(^{439}\)
6 If two elements compare as equal, their relative order in the resulting sorted array is unspecified.

\(^{437}\)In practice, this means that the entire array has been sorted according to the comparison function.
\(^{438}\)The `context` argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.
\(^{439}\)The `context` argument is for the use of the comparison function in performing its duties. For example, it might specify a collating sequence used by the comparison function.
Returns
7 The \texttt{qsort_s} function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.6.4 Multibyte/wide character conversion functions
1 The behavior of the multibyte character functions is affected by the \texttt{LC\_CTYPE} category of the current locale. For a state-dependent encoding, each function is placed into its initial conversion state by a call for which its character pointer argument, \( s \), is a null pointer. Subsequent calls with \( s \) as other than a null pointer cause the internal conversion state of the function to be altered as necessary. A call with \( s \) as a null pointer causes these functions to set the int pointed to by their \texttt{status} argument to a nonzero value if encodings have state dependency, and zero otherwise.\footnote{If the locale employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.} Changing the \texttt{LC\_CTYPE} category causes the conversion state of these functions to be indeterminate.

K.3.6.4.1 The \texttt{wctomb_s} function

Synopsis
1

\begin{verbatim}
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdlib.h>
errno_t wctomb_s(int *restrict status, char *restrict s, rsize_t smax, wchar_t wc);
\end{verbatim}

Runtime-constraints
2 Let \( n \) denote the number of bytes needed to represent the multibyte character corresponding to the wide character given by \( wc \) (including any shift sequences).
3 If \( s \) is not a null pointer, then \( smax \) shall not be less than \( n \), and \( smax \) shall not be greater than \texttt{FSIZE\_MAX}. If \( s \) is a null pointer, then \( smax \) shall equal zero.
4 If there is a runtime-constraint violation, \texttt{wctomb_s} does not modify the int pointed to by \texttt{status}, and if \( s \) is not a null pointer, no more than \( smax \) elements in the array pointed to by \( s \) will be accessed.

Description
5 The \texttt{wctomb_s} function determines \( n \) and stores the multibyte character representation of \( wc \) in the array whose first element is pointed to by \( s \) (if \( s \) is not a null pointer). The number of characters stored never exceeds \texttt{MB\_CUR\_MAX} or \( smax \). If \( wc \) is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state, and the function is left in the initial conversion state.
6 The implementation shall behave as if no library function calls the \texttt{wctomb_s} function.
7 If \( s \) is a null pointer, the \texttt{wctomb_s} function stores into the int pointed to by \texttt{status} a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings.
8 If \( s \) is not a null pointer, the \texttt{wctomb_s} function stores into the int pointed to by \texttt{status} either \( n \) or -1 if \( wc \), respectively, does or does not correspond to a valid multibyte character.
9 In no case will the int pointed to by \texttt{status} be set to a value greater than the \texttt{MB\_CUR\_MAX} macro.

Returns
10 The \texttt{wctomb_s} function returns zero if successful, and a nonzero value if there was a runtime-constraint violation or \( wc \) did not correspond to a valid multibyte character.

K.3.6.5 Multibyte/wide string conversion functions
1 The behavior of the multibyte string functions is affected by the \texttt{LC\_CTYPE} category of the current locale.

K.3.6.5.1 The \texttt{mbstowcs_s} function
Synopsis

```c
#include <stdlib.h>
errno_t mbstowcs_s(size_t * restrict retval, wchar_t * restrict dst, rsize_t dstmax, const char * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither `retval` nor `src` shall be a null pointer. If `dst` is not a null pointer, then neither `len` nor `dstmax` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer and `len` is not less than `dstmax`, then a null character shall occur within the first `dstmax` multibyte characters of the array pointed to by `src`.

3 If there is a runtime-constraint violation, then `mbstowcs_s` does the following. If `retval` is not a null pointer, then `mbstowcs_s` sets `*retval` to `(size_t)(-1)`. If `dst` is not a null pointer and `dstmax` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then `mbstowcs_s` sets `dst[0]` to the null wide character.

Description

4 The `mbstowcs_s` function converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by `src` into a sequence of corresponding wide characters. If `dst` is not a null pointer, the converted characters are stored into the array pointed to by `dst`. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if `dst` is not a null pointer) when `len` wide characters have been stored into the array pointed to by `dst`.\(^{441}\) If `dst` is not a null pointer and no null wide character was stored into the array pointed to by `dst`, then `dst[len]` is set to the null wide character. Each conversion takes place as if by a call to the `mbrtowc` function.

5 Regardless of whether `dst` is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the `mbstowcs_s` function stores the value `(size_t)(-1)` into `*retval`. Otherwise, the `mbstowcs_s` function stores into `*retval` the number of multibyte characters successfully converted, not including the terminating null character (if any).

6 All elements following the terminating null wide character (if any) written by `mbstowcs_s` in the array of `dstmax` wide characters pointed to by `dst` take unspecified values when `mbstowcs_s` returns.\(^{442}\)

7 If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

8 The `mbstowcs_s` function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.6.5.2 The `wcstombs_s` function

Synopsis

```c
#include <stdlib.h>
errno_t wcstombs_s(size_t * restrict retval, char * restrict dst, rsize_t dstmax, const wchar_t * restrict src, rsize_t len);
```

Runtime-constraints

2 Neither `retval` nor `src` shall be a null pointer. If `dst` is not a null pointer, then `len` shall not be greater than `RSIZE_MAX/sizeof(wchar_t)` and `dstmax` shall be nonzero and not greater than `RSIZE_MAX`. If `dst` is a null pointer, then `dstmax` shall equal zero. If `dst` is not a null pointer and

\(^{441}\)Thus, the value of `len` is ignored if `dst` is a null pointer.

\(^{442}\)This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.
len is not less than dstmax, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

If there is a runtime-constraint violation, then wcstombs_s does the following. If retval is not a null pointer, then wcstombs_s sets *retval to (size_t)(-1). If dst is not a null pointer and dstmax is greater than zero and not greater than RSIZE_MAX, then wcstombs_s sets dst[0] to the null character.

Description

The wcstombs_s function converts a sequence of wide characters from the array pointed to by src into a sequence of corresponding multibyte characters that begins in the initial shift state. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

- when a wide character is reached that does not correspond to a valid multibyte character;
- (if dst is not a null pointer) when the next multibyte character would exceed the limit of n total bytes to be stored into the array pointed to by dst. If the wide character being converted is the null wide character, then n is the lesser of len or dstmax. Otherwise, n is the lesser of len or dstmax-1.

If the conversion stops without converting a null wide character and dst is not a null pointer, then a null character is stored into the array pointed to by dst immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the wcrtomb function.

Regardless of whether dst is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the wcstombs_s function stores the value (size_t)(-1) into *retval. Otherwise, the wcstombs_s function stores into *retval the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

All elements following the terminating null character (if any) written by wcstombs_s in the array of dstmax elements pointed to by dst take unspecified values when wcstombs_s returns.

If copying takes place between objects that overlap, the objects take on unspecified values.

Returns

The wcstombs_s function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.7 String handling <string.h>

The header <string.h> defines two types.

The types are

```c
errno_t
```

which is type int; and

```c
rsize_t
```

which is the type size_t.

K.3.7.1 Copying functions

K.3.7.1.1 The memcpy_s function

443)If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.

444)When len is not less than dstmax, the implementation might fill the array before discovering a runtime-constraint violation.
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memcpy_s(void * restrict s1, rsize_t s1max, const void * restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither s1 nor s2 shall be a null pointer. Neither s1max nor n shall be greater than RSIZE_MAX. n shall not be greater than s1max. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, the memcpy_s function stores zeros in the first s1max characters of the object pointed to by s1 if s1 is not a null pointer and s1max is not greater than RSIZE_MAX.

Description

4 The memcpy_s function copies n characters from the object pointed to by s2 into the object pointed to by s1.

Returns

5 The memcpy_s function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.2 The memmove_s function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memmove_s(void *s1, rsize_t s1max, const void *s2, rsize_t n);
```

Runtime-constraints

2 Neither s1 nor s2 shall be a null pointer. Neither s1max nor n shall be greater than RSIZE_MAX. n shall not be greater than s1max.

3 If there is a runtime-constraint violation, the memmove_s function stores zeros in the first s1max characters of the object pointed to by s1 if s1 is not a null pointer and s1max is not greater than RSIZE_MAX.

Description

4 The memmove_s function copies n characters from the object pointed to by s2 into the object pointed to by s1. This copying takes place as if the n characters from the object pointed to by s2 are first copied into a temporary array of n characters that does not overlap the objects pointed to by s1 or s2, and then the n characters from the temporary array are copied into the object pointed to by s1.

Returns

5 The memmove_s function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.
K.3.7.1.3 The \texttt{strcpy\_s} function

Synopsis

\begin{verbatim}
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strcpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
\end{verbatim}

Runtime-constraints

2 Neither \texttt{s1} nor \texttt{s2} shall be a null pointer. \texttt{s1max} shall not be greater than \texttt{RSIZE\_MAX}. \texttt{s1max} shall not equal zero. \texttt{s1max} shall be greater than \texttt{strnlen\_s(s2, s1max)}. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if \texttt{s1} is not a null pointer and \texttt{s1max} is greater than zero and not greater than \texttt{RSIZE\_MAX}, then \texttt{strcpy\_s} sets \texttt{s1[0]} to the null character.

Description

4 The \texttt{strcpy\_s} function copies the string pointed to by \texttt{s2} (including the terminating null character) into the array pointed to by \texttt{s1}.

5 All elements following the terminating null character (if any) written by \texttt{strcpy\_s} in the array of \texttt{s1max} characters pointed to by \texttt{s1} take unspecified values when \texttt{strcpy\_s} returns.\footnote{This allows an implementation to copy characters from \texttt{s2} to \texttt{s1} while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of \texttt{s1} before discovering that the first element was set to the null character.}

Returns

6 The \texttt{strcpy\_s} function returns zero\footnote{A zero return value implies that all of the requested characters from the string pointed to by \texttt{s2} fit within the array pointed to by \texttt{s1} and that the result in \texttt{s1} is null terminated.} if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.1.4 The \texttt{strncpy\_s} function

Synopsis

\begin{verbatim}
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncpy_s(char * restrict s1, rsize_t s1max, const char * restrict s2, rsize_t n);
\end{verbatim}

Runtime-constraints

2 Neither \texttt{s1} nor \texttt{s2} shall be a null pointer. Neither \texttt{s1max} nor \texttt{n} shall be greater than \texttt{RSIZE\_MAX}. \texttt{s1max} shall not equal zero. If \texttt{n} is not less than \texttt{s1max}, then \texttt{s1max} shall be greater than \texttt{strnlen\_s(s2, s1max)}. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, then if \texttt{s1} is not a null pointer and \texttt{s1max} is greater than zero and not greater than \texttt{RSIZE\_MAX}, then \texttt{strncpy\_s} sets \texttt{s1[0]} to the null character.

Description

4 The \texttt{strncpy\_s} function copies not more than \texttt{n} successive characters (characters that follow a null character are not copied) from the array pointed to by \texttt{s2} to the array pointed to by \texttt{s1}. If no null character was copied from \texttt{s2}, then \texttt{s1[n]} is set to a null character.

5 All elements following the terminating null character (if any) written by \texttt{strncpy\_s} in the array of \texttt{s1max} characters pointed to by \texttt{s1} take unspecified values when \texttt{strncpy\_s} returns a nonzero value.\footnote{This allows an implementation to copy characters from \texttt{s2} to \texttt{s1} while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of \texttt{s1} before discovering that the first element was set to the null character.}

\footnote{This allows an implementation to copy characters from \texttt{s2} to \texttt{s1} while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of \texttt{s1} before discovering that the first element was set to the null character.}
6 The strncpy_s function returns zero\(^{448}\) if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

7 **EXAMPLE 1** The strncpy_s function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

/* ...
 */
char src1[100] = "hello";
char src2[7] = {'g', 'o', 'o', 'd', 'b', 'y', 'e'};
char dst1[6], dst2[5], dst3[5];
int r1, r2, r3;

r1 = strncpy_s(dst1, 6, src1, 100);
r2 = strncpy_s(dst2, 5, src2, 7);
r3 = strncpy_s(dst3, 5, src2, 4);
```

The first call will assign to r1 the value zero and to dst1 the sequence hello\0.
The second call will assign to r2 a nonzero value and to dst2 the sequence \0.
The third call will assign to r3 the value zero and to dst3 the sequence good\0.

K.3.7.2 Concatenation functions
K.3.7.2.1 The strcat_s function

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

errno_t strcat_s(char * restrict s1, rsize_t s1max, const char * restrict s2);
```

**Runtime-constraints**

2 Let \( m \) denote the value \( s1max - \text{strnlen_s}(s1, s1max) \) upon entry to strcat_s.

3 Neither \( s1 \) nor \( s2 \) shall be a null pointer. \( s1max \) shall not be greater than \( \text{RSIZE_MAX} \). \( s1max \) shall not equal zero. \( m \) shall not equal zero.\(^{449}\) \( m \) shall be greater than \( \text{strnlen_s}(s2, m) \). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if \( s1 \) is not a null pointer and \( s1max \) is greater than zero and not greater than \( \text{RSIZE_MAX} \), then strcat_s sets \( s1[0] \) to the null character.

**Description**

5 The strcat_s function appends a copy of the string pointed to by \( s2 \) (including the terminating null character) to the end of the string pointed to by \( s1 \). The initial character from \( s2 \) overwrites the null character at the end of \( s1 \).

6 All elements following the terminating null character (if any) written by strcat_s in the array of \( s1max \) characters pointed to by \( s1 \) take unspecified values when strcat_s returns.\(^{450}\)

**Returns**

7 The strcat_s function returns zero\(^{451}\) if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

---

448) A zero return value implies that all of the requested characters from the string pointed to by \( s2 \) fit within the array pointed to by \( s1 \) and that the result in \( s1 \) is null terminated.

449) Zero means that \( s1 \) was not null terminated upon entry to strcat_s.

450) This allows an implementation to append characters from \( s2 \) to \( s1 \) while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of \( s1 \) before discovering that the first element was set to the null character.

451) A zero return value implies that all of the requested characters from the string pointed to by \( s2 \) were appended to the string pointed to by \( s1 \) and that the result in \( s1 \) is null terminated.
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strncat_s(char * restrict s1, rsize_t s1max, const char * restrict s2, rsize_t n);
```

Runtime-constraints

2 Let $m$ denote the value $s1max - strnlen_s(s1, s1max)$ upon entry to `strncat_s`.

3 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX`. `s1max` shall not equal zero. $m$ shall not equal zero.

4 If there is a runtime-constraint violation, then if `s1` is not a null pointer and `s1max` is greater than zero and not greater than `RSIZE_MAX`, then `strncat_s` sets `s1[0]` to the null character.

Description

5 The `strncat_s` function appends not more than `n` successive characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character from `s2` overwrites the null character at the end of `s1`. If no null character was copied from `s2`, then `s1[s1max - m + n]` is set to a null character.

6 All elements following the terminating null character (if any) written by `strncat_s` in the array of `s1max` characters pointed to by `s1` take unspecified values when `strncat_s` returns.

Returns

7 The `strncat_s` function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

8 **EXAMPLE 1** The `strncat_s` function can be used to copy a string without the danger that the result will not be null terminated or that characters will be written past the end of the destination array.

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
/* ... */
char s1[100] = "good";
char s2[6] = "hello";
char s3[6] = "hello";
char s4[7] = "abc";
char s5[1000] = "bye";
int r1, r2, r3, r4;
  r1 = strncat_s(s1, 100, s5, 1000);
  r2 = strncat_s(s2, 6, "", 1);
  r3 = strncat_s(s3, 6, "X", 2);
  r4 = strncat_s(s4, 7, "defghijklm", 3);
```

After the first call `r1` will have the value zero and `s1` will contain the sequence `goodbye\0`.

After the second call `r2` will have the value zero and `s2` will contain the sequence `hello\0`.

After the third call `r3` will have a nonzero value and `s3` will contain the sequence `\0`.

After the fourth call `r4` will have the value zero and `s4` will contain the sequence `abcdef\0`.

K.3.7.3 Search functions

K.3.7.3.1 The `strtok_s` function

[452]Zero means that `s1` was not null terminated upon entry to `strncat_s`.

[453]This allows an implementation to append characters from `s2` to `s1` while simultaneously checking if any of those characters are null. Such an approach might write a character to every element of `s1` before discovering that the first element was set to the null character.

[454]A zero return value implies that all of the requested characters from the string pointed to by `s2` were appended to the string pointed to by `s1` and that the result in `s1` is null terminated.
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
char *strtok_s(char * restrict s1, rsize_t * restrict s1max,
               const char * restrict s2, char ** restrict ptr);
```

Runtime-constraints

2 None of `s1max`, `s2`, or `ptr` shall be a null pointer. If `s1` is a null pointer, then `*ptr` shall not be a null pointer. The value of `*s1max` shall not be greater than `RSIZE_MAX`. The end of the token found shall occur within the first `*s1max` characters of `s1` for the first call, and shall occur within the first `*s1max` characters of where searching resumes on subsequent calls.

3 If there is a runtime-constraint violation, the `strtok_s` function does not indirect through the `s1` or `s2` pointers, and does not store a value in the object pointed to by `ptr`.

Description

4 A sequence of calls to the `strtok_s` function breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a character from the string pointed to by `s2`. The fourth argument points to a caller-provided `char` pointer into which the `strtok_s` function stores information necessary for it to continue scanning the same string.

5 The first call in a sequence has a non-null first argument and `s1max` points to an object whose value is the number of elements in the character array pointed to by the first argument. The first call stores an initial value in the object pointed to by `ptr` and updates the value pointed to by `s1max` to reflect the number of elements that remain in relation to `ptr`. Subsequent calls in the sequence have a null first argument and the objects pointed to by `s1max` and `ptr` are required to have the values stored by the previous call in the sequence, which are then updated. The separator string pointed to by `s2` may be different from call to call.

6 The first call in the sequence searches the string pointed to by `s1` for the first character that is not contained in the current separator string pointed to by `s2`. If no such character is found, then there are no tokens in the string pointed to by `s1` and the `strtok_s` function returns a null pointer. If such a character is found, it is the start of the first token.

7 The `strtok_s` function then searches from there for the first character in `s1` that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches in the same string for a token return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token.

8 In all cases, the `strtok_s` function stores sufficient information in the pointer pointed to by `ptr` so that subsequent calls, with a null pointer for `s1` and the unmodified pointer value for `ptr`, shall start searching just past the element overwritten by a null character (if any).

Returns

9 The `strtok_s` function returns a pointer to the first character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

EXAMPLE

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
static char str1[] = "?a???b,,,#c";
static char str2[] = "\t \t";
char *t, *ptr1, *ptr2;
rsize_t max1 = sizeof (str1);
rsize_t max2 = sizeof (str2);

int main(void)
{   t = strtok_s(str1, &max1, ",", &ptr1);  // t points to the token "a"
    t = strtok_s(NULL, &max1, ",", &ptr1);  // t points to the token "??b"
    t = strtok_s(str2, &max2, " \t", &ptr2);  // t is a null pointer
    t = strtok_s(NULL, &max1, "#,", &ptr1);  // t points to the token "c"
    return 0;
}
```
K.3.7.4 Miscellaneous functions
K.3.7.4.1 The \texttt{memset\_s} function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n);
```

Runtime-constraints

2 \texttt{s} shall not be a null pointer. Neither \texttt{smax} nor \texttt{n} shall be greater than \texttt{RSIZE\_MAX}. \texttt{n} shall not be greater than \texttt{smax}.

3 If there is a runtime-constraint violation, then if \texttt{s} is not a null pointer and \texttt{smax} is not greater than \texttt{RSIZE\_MAX}, the \texttt{memset\_s} function stores the value of \texttt{c} (converted to an \texttt{unsigned char}) into each of the first \texttt{smax} characters of the object pointed to by \texttt{s}.

Description

4 The \texttt{memset\_s} function copies the value of \texttt{c} (converted to an \texttt{unsigned char}) into each of the first \texttt{n} characters of the object pointed to by \texttt{s}. Unlike \texttt{memset}, any call to the \texttt{memset\_s} function shall be evaluated strictly according to the rules of the abstract machine as described in (5.1.2.3). That is, any call to the \texttt{memset\_s} function shall assume that the memory indicated by \texttt{s} and \texttt{n} may be accessible in the future and thus contains the values indicated by \texttt{c}.

Returns

5 The \texttt{memset\_s} function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.7.4.2 The \texttt{strerror\_s} function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
errno_t strerror_s(char *, rsize_t maxsize, errno_t errnum);
```

Runtime-constraints

2 \texttt{s} shall not be a null pointer. \texttt{maxsize} shall not be greater than \texttt{RSIZE\_MAX}. \texttt{maxsize} shall not equal zero.

3 If there is a runtime-constraint violation, then the array (if any) pointed to by \texttt{s} is not modified.

Description

4 The \texttt{strerror\_s} function maps the number in \texttt{errnum} to a locale-specific message string. Typically, the values for \texttt{errnum} come from \texttt{errno}, but \texttt{strerror\_s} shall map any value of type \texttt{int} to a message.

5 If the length of the desired string is less than \texttt{maxsize}, then the string is copied to the array pointed to by \texttt{s}.

6 Otherwise, if \texttt{maxsize} is greater than zero, then \texttt{maxsize-1} characters are copied from the string to the array pointed to by \texttt{s} and then \texttt{s[maxsize-1]} is set to the null character. Then, if \texttt{maxsize} is greater than 3, then \texttt{s[maxsize-2]}, \texttt{s[maxsize-3]}, and \texttt{s[maxsize-4]} are set to the character period (\texttt{.}).

Returns

7 The \texttt{strerror\_s} function returns zero if the length of the desired string was less than \texttt{maxsize} and there was no runtime-constraint violation. Otherwise, the \texttt{strerror\_s} function returns a nonzero value.
K.3.7.4.3 The `strerrorlen_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strerrorlen_s(errno_t errnum);
```

Description

The `strerrorlen_s` function calculates the length of the (untruncated) locale-specific message string that the `strerror_s` function maps to `errnum`.

Returns

The `strerrorlen_s` function returns the number of characters (not including the null character) in the full message string.

K.3.7.4.4 The `strnlen_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>
size_t strnlen_s(const char *s, size_t maxsize);
```

Description

The `strnlen_s` function computes the length of the string pointed to by `s`.

Returns

1. If `s` is a null pointer, the `strnlen_s` function returns zero.
2. Otherwise, the `strnlen_s` function returns the number of characters that precede the terminating null character. If there is no null character in the first `maxsize` characters of `s` then `strnlen_s` returns `maxsize`. At most the first `maxsize` characters of `s` shall be accessed by `strnlen_s`.

K.3.8 Date and time `<time.h>`

The header `<time.h>` defines two types.

The types are

```
errno_t
```

which is type `int`; and

```
rsize_t
```

which is the type `size_t`.

K.3.8.1 Components of time

A broken-down time is normalized if the values of the members of the `tm` structure are in their normal ranges.

K.3.8.2 Time conversion functions

Like the `strftime` function, the `asctime_s` and `ctime_s` functions do not return a pointer to a static object, and other library functions are permitted to call them.

K.3.8.2.1 The `asctime_s` function

455) Note that the `strnlen_s` function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated string argument make `strnlen_s` useful in algorithms that gracefully handle such exceptional data.

456) The normal ranges are defined in 7.27.1.
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t asctime_s(char *s, rsize_t maxsize, const struct tm *timeptr);
```

Runtime-constraints

Neither `s` nor `timeptr` shall be a null pointer. `maxsize` shall not be less than 26 and shall not be greater than `RSIZE_MAX`. The broken-down time pointed to by `timeptr` shall be normalized. The calendar year represented by the broken-down time pointed to by `timeptr` shall not be less than calendar year 0 and shall not be greater than calendar year 9999.

If there is a runtime-constraint violation, there is no attempt to convert the time, and `s[0]` is set to a null character if `s` is not a null pointer and `maxsize` is not zero and is not greater than `RSIZE_MAX`.

Description

The `asctime_s` function converts the normalized broken-down time in the structure pointed to by `timeptr` into a 26 character (including the null character) string in the form

```
Sun Sep 16 01:03:52 1973
```

The fields making up this string are (in order):

1. The name of the day of the week represented by `timeptr->tm_wday` using the following three character weekday names: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
2. The character space.
3. The name of the month represented by `timeptr->tm_mon` using the following three character month names: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
4. The character space.
5. The value of `timeptr->tm_mday` as if printed using the `fprintf` format "%2d".
6. The character space.
7. The value of `timeptr->tm_hour` as if printed using the `fprintf` format "%.2d".
8. The character colon.
9. The value of `timeptr->tm_min` as if printed using the `fprintf` format "%.2d".
10. The character colon.
11. The value of `timeptr->tm_year + 1900` as if printed using the `fprintf` format "%4d".
12. The character new line.
13. The null character.

Recommended practice

The `strftime` function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the `asctime_s` function, consider using the `strftime` function instead.

Returns

The `asctime_s` function returns zero if the time was successfully converted and stored into the array pointed to by `s`. Otherwise, it returns a nonzero value.
K.3.8.2.2 The \texttt{ctime\_s} function

Synopsis

\begin{verbatim}
define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
errno_t ctime_s(char *s, rsize_t maxsize, const time_t *timer);
\end{verbatim}

Runtime-constraints
2 Neither \texttt{s} nor \texttt{timer} shall be a null pointer. \texttt{maxsize} shall not be less than 26 and shall not be greater than \texttt{RSIZE\_MAX}.

3 If there is a runtime-constraint violation, \texttt{s[0]} is set to a null character if \texttt{s} is not a null pointer and \texttt{maxsize} is not equal zero and is not greater than \texttt{RSIZE\_MAX}.

Description
4 The \texttt{ctime\_s} function converts the calendar time pointed to by \texttt{timer} to local time in the form of a string. It is equivalent to

\begin{verbatim}
asctime_s(s, maxsize, localtime_s(timer, &(struct tm){ 0 }));
\end{verbatim}

Recommended practice
The \texttt{strftime} function allows more flexible formatting and supports locale-specific behavior. If you do not require the exact form of the result string produced by the \texttt{ctime\_s} function, consider using the \texttt{strftime} function instead.

Returns
5 The \texttt{ctime\_s} function returns zero if the time was successfully converted and stored into the array pointed to by \texttt{s}. Otherwise, it returns a nonzero value.

K.3.8.2.3 The \texttt{gmtime\_s} function

Synopsis

\begin{verbatim}
define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *gmtime_s(const time_t *restrict timer, struct tm *restrict result);
\end{verbatim}

Runtime-constraints
2 Neither \texttt{timer} nor \texttt{result} shall be a null pointer.

3 If there is a runtime-constraint violation, there is no attempt to convert the time.

Description
4 The \texttt{gmtime\_s} function converts the calendar time pointed to by \texttt{timer} into a broken-down time, expressed as UTC. The broken-down time is stored in the structure pointed to by \texttt{result}.

Returns
5 The \texttt{gmtime\_s} function returns \texttt{result}, or a null pointer if the specified time cannot be converted to UTC or there is a runtime-constraint violation.

K.3.8.2.4 The \texttt{localtime\_s} function

Synopsis

\begin{verbatim}
define __STDC_WANT_LIB_EXT1__ 1
#include <time.h>
struct tm *localtime_s(const time_t *restrict timer, struct tm *restrict result);
\end{verbatim}

Runtime-constraints
2 Neither \texttt{timer} nor \texttt{result} shall be a null pointer.
If there is a runtime-constraint violation, there is no attempt to convert the time.

Description

The `localtime_s` function converts the calendar time pointed to by `timer` into a broken-down time, expressed as local time. The broken-down time is stored in the structure pointed to by `result`.

Returns

The `localtime_s` function returns `result`, or a null pointer if the specified time cannot be converted to local time or there is a runtime-constraint violation.

K.3.9 Extended multibyte and wide character utilities `<wchar.h>`

The header `<wchar.h>` defines two types.

- `errno_t` which is type `int`;
- `rsize_t` which is the type `size_t`.

Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the objects take on unspecified values.

K.3.9.1 Formatted wide character input/output functions

K.3.9.1.1 The `fwprintf_s` function

Synopsis

```c
#include <wchar.h>

int fwprintf_s(FILE * restrict stream, const wchar_t * restrict format, ...);
```

Runtime-constraints

Neither `stream` nor `format` shall be a null pointer. The `%n` specifier\(^{457}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `fwprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

If there is a runtime-constraint violation, the `fwprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `fwprintf_s` produced output before discovering the runtime-constraint violation.

Description

The `fwprintf_s` function is equivalent to the `fwprintf` function except for the explicit runtime-constraints listed above.

Returns

The `fwprintf_s` function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.2 The `fwscanf_s` function

Synopsis

```c
#include <stdio.h>

int fwscanf_s(FILE * restrict stream, wchar_t * restrict format, ...);
```

\(^{457}\)It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%n"`. 
#include <wchar.h>
int _fwscanf_s(FILE * restrict stream, const wchar_t * restrict format, ...);

Runtime-constraints

Neither stream nor format shall be a null pointer. Any argument indirection though in order to store converted input shall not be a null pointer.

If there is a runtime-constraint violation, the _fwscanf_s function does not attempt to perform further input, and it is unspecified to what extent _fwscanf_s performed input before discovering the runtime-constraint violation.

Description

The _fwscanf_s function is equivalent to _fwscanf except that the c, s, and [ conversion specifiers apply to a pair of arguments (unless assignment suppression is indicated by a *). The first of these arguments is the same as for _fwscanf. That argument is immediately followed in the argument list by the second argument, which has type _size_t and gives the number of elements in the array pointed to by the first argument of the pair. If the first argument points to a scalar object, it is considered to be an array of one element.\(^{458}\)

A matching failure occurs if the number of elements in a receiving object is insufficient to hold the converted input (including any trailing null character).

Returns

The _fwscanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the _fwscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.3 The _snwprintf_s function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int _snwprintf_s(wchar_t * restrict s, _rsize_t n, const wchar_t * restrict format, ...);
```

Runtime-constraints

Neither s nor format shall be a null pointer. n shall neither equal zero nor be greater than RSIZE_MAX/sizeof(wchar_t). The %n specifier\(^{459}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by format. Any argument to _snwprintf_s corresponding to a %s specifier shall not be a null pointer. No encoding error shall occur.

If there is a runtime-constraint violation, then if s is not a null pointer and n is greater than zero and not greater than RSIZE_MAX/sizeof(wchar_t), then the _snwprintf_s function sets s[0] to the null wide character.

Description

The _snwprintf_s function is equivalent to the _swprintf function except for the explicit runtime-constraints listed above.

\(^{458}\)If the format is known at translation time, an implementation can issue a diagnostic for any argument used to store the result from a c, s, or [ conversion specifier if that argument is not followed by an argument of a type compatible with _size_t. A limited amount of checking can be done if even if the format is not known at translation time. For example, an implementation could issue a diagnostic for each argument after format that has of type pointer to one of char, signed char, unsigned char, or void that is not followed by an argument of a type compatible with _size_t. The diagnostic could warn that unless the pointer is being used with a conversion specifier using the hh length modifier, a length argument is expected to follow the pointer argument. Another useful diagnostic could flag any non-pointer argument following format that did not have a type compatible with _size_t.

\(^{459}\)It is not a runtime-constraint violation for the wide characters %n to appear in sequence in the wide string pointed at by format when those wide characters are not a interpreted as a %n specifier. For example, if the entire format string was L"%nnn".
The `snwprintf_s` function, unlike `swprintf_s`, will truncate the result to fit within the array pointed to by `s`.

**Returns**

The `snwprintf_s` function returns the number of wide characters that would have been written had `n` been sufficiently large, not counting the terminating wide null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than `n`.

K.3.9.1.4 The `swprintf_s` function

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int swprintf_s(wchar_t * restrict s, rsize_t n, const wchar_t * restrict format, ...);
```

**Runtime-constraints**

Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX/sizeof(wchar_t)`. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by `s` shall not be greater than `n`. The `%n` specifier (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `swprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

No encoding error shall occur.

If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `swprintf_s` function sets `s[0]` to the null wide character.

**Description**

The `swprintf_s` function is equivalent to the `swprintf` function except for the explicit runtime-constraints listed above.

The `swprintf_s` function, unlike `snwprintf_s`, treats a result too big for the array pointed to by `s` as a runtime-constraint violation.

**Returns**

If no runtime-constraint violation occurred, the `swprintf_s` function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if `n` or more wide characters are requested to be written, `swprintf_s` returns a negative value. If any other runtime-constraint violation occurred, `swprintf_s` returns zero.

K.3.9.1.5 The `swscanf_s` function

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int swscanf_s(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

**Runtime-constraints**

Neither `s` nor `format` shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

If there is a runtime-constraint violation, the `swscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `swscanf_s` performed input before discovering the runtime-constraint violation.

It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not interpreted as a `%n` specifier. For example, if the entire format string was L"%%%n".
Description
4 The `swscanf_s` function is equivalent to `fwscanf_s`, except that the argument `s` specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the `fwscanf_s` function.

Returns
5 The `swscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `swscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.6 The `vfwprintf_s` function

Synopsis
1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

Runtime-constraints
2 Neither `stream` nor `format` shall be a null pointer. The `%n` specifier\(^{[46]}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vfwprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vfwprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `vfwprintf_s` produced output before discovering the runtime-constraint violation.

Description
4 The `vfwprintf_s` function is equivalent to the `vfwprintf` function except for the explicit runtime-constraints listed above.

Returns
5 The `vfwprintf_s` function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.7 The `vfscanf_s` function

Synopsis
1
```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfscanf_s(FILE * restrict stream, const wchar_t * restrict format, va_list arg);
```

Runtime-constraints
2 Neither `stream` nor `format` shall be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vfscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `vfscanf_s` performed input before discovering the runtime-constraint violation.

\(^{[46]}\)It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was L"L"L"L"L":

556 Bounds-checking interfaces § K.3.9.1.7
Description

4 The `vfwscanf_s` function is equivalent to `fwscanf_s`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfwscanf_s` function does not invoke the `va_end` macro.\(^{462}\)

Returns

5 The `vfwscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vfwscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.8 The `vsnwprintf_s` function

Synopsis

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vsnwprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, va_list arg);
```

Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX/sizeof(wchar_t)`. The `%n` specifier\(^{463}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vsnwprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `vsnwprintf_s` function sets `s[0]` to the null wide character.

Description

4 The `vsnwprintf_s` function is equivalent to the `vswprintf` function except for the explicit runtime-constraints listed above.

5 The `vsnwprintf_s` function, unlike `vswprintf_s`, will truncate the result to fit within the array pointed to by `s`.

Returns

6 The `vsnwprintf_s` function returns the number of wide characters that would have been written had `n` been sufficiently large, not counting the terminating null character, or a negative value if a runtime-constraint violation occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than `n`.

K.3.9.1.9 The `vswprintf_s` function

Synopsis

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswprintf_s(wchar_t *restrict s, rsize_t n, const wchar_t *restrict format, va_list arg);
```

\(^{462}\)As the functions `vfwscanf_s`, `vwscanf_s`, and `vswscanf_s` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.

\(^{463}\)It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was L"\%n".
### Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. `n` shall neither equal zero nor be greater than `RSIZE_MAX/sizeof(wchar_t)`. The number of wide characters (including the trailing null) required for the result to be written to the array pointed to by `s` shall not be greater than `n`. The `%n` specifier (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vswprintf_s` corresponding to a `%s` specifier shall not be a null pointer. No encoding error shall occur.

3 If there is a runtime-constraint violation, then if `s` is not a null pointer and `n` is greater than zero and not greater than `RSIZE_MAX/sizeof(wchar_t)`, then the `vswprintf_s` function sets `s[0]` to the null wide character.

### Description

4 The `vswprintf_s` function is equivalent to the `vswprintf` function except for the explicit runtime-constraints listed above.

5 The `vswprintf_s` function, unlike `vsnwprintf_s`, treats a result too big for the array pointed to by `s` as a runtime-constraint violation.

### Returns

6 If no runtime-constraint violation occurred, the `vswprintf_s` function returns the number of wide characters written in the array, not counting the terminating null wide character. If an encoding error occurred or if `n` or more wide characters are requested to be written, `vswprintf_s` returns a negative value. If any other runtime-constraint violation occurred, `vswprintf_s` returns zero.

### Synopsis

```c
#define  __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vswprintf_s(const wchar_t * restrict s, const wchar_t * restrict format, va_list arg);
```

### Runtime-constraints

2 Neither `s` nor `format` shall be a null pointer. Any argument indirection though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the `vswscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `vswscanf_s` performed input before discovering the runtime-constraint violation.

### Description

4 The `vswscanf_s` function is equivalent to `swscanf_s`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vswscanf_s` function does not invoke the `va_end` macro.

### Returns

5 The `vswscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vswscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

### Notes

644 It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

645 As the functions `vfscanf_s`, `vscanf_s`, and `vswscanf_s` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.
Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwprintf_s(const wchar_t * restrict format, va_list arg);
```

Runtime-constraints

1. `format` shall not be a null pointer. The `%n` specifier\(^{466}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by `format`. Any argument to `vwprintf_s` corresponding to a `%s` specifier shall not be a null pointer.

2. If there is a runtime-constraint violation, the `vwprintf_s` function does not attempt to produce further output, and it is unspecified to what extent `vwprintf_s` produced output before discovering the runtime-constraint violation.

Description

3. The `vwprintf_s` function is equivalent to the `vwprintf` function except for the explicit runtime-constraints listed above.

Returns

4. The `vwprintf_s` function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

K.3.9.1.12 The `vwscanf_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vwscanf_s(const wchar_t * restrict format, va_list arg);
```

Runtime-constraints

1. `format` shall not be a null pointer. Any argument indrected though in order to store converted input shall not be a null pointer.

2. If there is a runtime-constraint violation, the `vwscanf_s` function does not attempt to perform further input, and it is unspecified to what extent `vwscanf_s` performed input before discovering the runtime-constraint violation.

Description

3. The `vwscanf_s` function is equivalent to `wscanf_s`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vwscanf_s` function does not invoke the `va_end` macro.\(^{467}\)

Returns

4. The `vwscanf_s` function returns the value of the macro `EOF` if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the `vwscanf_s` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.1.13 The `vprintf_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdarg.h>
#include <wchar.h>
int vprintf_s(const wchar_t * restrict format, va_list arg);
```

\(^{466}\)It is not a runtime-constraint violation for the wide characters `%n` to appear in sequence in the wide string pointed at by `format` when those wide characters are not a interpreted as a `%n` specifier. For example, if the entire format string was `L"%%n"`.

\(^{467}\)As the functions `vfscanf_s`, `vwscanf_s`, and `vswscanf_s` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.

§ K.3.9.1.13 Bounds-checking interfaces 559
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wprintf_s(const wchar_t * restrict format, ...);

Runtime-constraints
2 format shall not be a null pointer. The %n specifier\(^{685}\) (modified or not by flags, field width, or precision) shall not appear in the wide string pointed to by format. Any argument to wprintf_s corresponding to a %s specifier shall not be a null pointer.

3 If there is a runtime-constraint violation, the wprintf_s function does not attempt to produce further output, and it is unspecified to what extent wprintf_s produced output before discovering the runtime-constraint violation.

Description
4 The wprintf_s function is equivalent to the wprintf function except for the explicit runtime-constraints listed above.

Returns
5 The wprintf_s function returns the number of wide characters transmitted, or a negative value if an output error, encoding error, or runtime-constraint violation occurred.

\^{685}\)It is not a runtime-constraint violation for the wide characters %n to appear in sequence in the wide string pointed at by format when those wide characters are not a interpreted as a %n specifier. For example, if the entire format string was L"%nnn".

K.3.9.1.14 The wscanf_s function

Synopsis
1 #define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
int wscanf_s(const wchar_t * restrict format, ...);

Runtime-constraints
2 format shall not be a null pointer. Any argument indirected though in order to store converted input shall not be a null pointer.

3 If there is a runtime-constraint violation, the wscanf_s function does not attempt to perform further input, and it is unspecified to what extent wscanf_s performed input before discovering the runtime-constraint violation.

Description
4 The wscanf_s function is equivalent to fwscanf_s with the argument stdin interposed before the arguments to wscanf_s.

Returns
5 The wscanf_s function returns the value of the macro EOF if an input failure occurs before any conversion or if there is a runtime-constraint violation. Otherwise, the wscanf_s function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

K.3.9.2 General wide string utilities
K.3.9.2.1 Wide string copying functions
K.3.9.2.1.1 The wcscpy_s function

Synopsis
1 #define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcscpy_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
Runtime-constraints

Neither \( s_1 \) nor \( s_2 \) shall be a null pointer. \( s_{1\text{max}} \) shall not be greater than \( \text{sizeof}(\text{wchar}_t) \). \( s_{1\text{max}} \) shall not equal zero. \( s_{1\text{max}} \) shall be greater than \( \text{wcsnlen}_s(s_2, s_{1\text{max}}) \). Copying shall not take place between objects that overlap.

If there is a runtime-constraint violation, then if \( s_1 \) is not a null pointer and \( s_{1\text{max}} \) is greater than zero and not greater than \( \text{sizeof}(\text{wchar}_t) \), then \( \text{wcsncpy}_s \) sets \( s_1[0] \) to the null wide character.

Description

The \( \text{wcsncpy}_s \) function copies not more than \( n \) successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by \( s_2 \) to the array pointed to by \( s_1 \). If no null wide character was copied from \( s_2 \), then \( s_1[n] \) is set to a null wide character.

All elements following the terminating null wide character (if any) written by \( \text{wcsncpy}_s \) in the array of \( s_{1\text{max}} \) wide characters pointed to by \( s_1 \) take unspecified values when \( \text{wcsncpy}_s \) returns.

Returns

The \( \text{wcsncpy}_s \) function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2.1 Bounds-checking interfaces
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>

/* ... */
wchar_t src1[100] = L"hello";
wchar_t src2[7] = {L'g', L'o', L'o', L'd', L'b', L'y', L'e'};
wchar_t dst1[6], dst2[5], dst3[5];
int r1, r2, r3;
    r1 = wcsncpy_s(dst1, 6, src1, 100);
    r2 = wcsncpy_s(dst2, 5, src2, 7);
    r3 = wcsncpy_s(dst3, 5, src2, 4);

The first call will assign to `r1` the value zero and to `dst1` the sequence of wide characters `hello\0`.
The second call will assign to `r2` a nonzero value and to `dst2` the sequence of wide characters `\0`.
The third call will assign to `r3` the value zero and to `dst3` the sequence of wide characters `good\0`.

K.3.9.2.1.3 The `wcsncpy_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wcsncpy_s(const wchar_t *restrict s1, rsize_t s1max, const wchar_t *restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. `n` shall not be greater than `s1max`. Copying shall not take place between objects that overlap.

3 If there is a runtime-constraint violation, the `wcsncpy_s` function stores zeros in the first `s1max` wide characters of the object pointed to by `s1` if `s1` is not a null pointer and `s1max` is not greater than `RSIZE_MAX/sizeof(wchar_t)`.

Description

4 The `wcsncpy_s` function copies `n` successive wide characters from the object pointed to by `s2` into the object pointed to by `s1`.

Returns

5 The `wcsncpy_s` function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.1.4 The `wmemmove_s` function

Synopsis

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
errno_t wmemmove_s(wchar_t *restrict s1, rsize_t s1max, const wchar_t *restrict s2, rsize_t n);
```

Runtime-constraints

2 Neither `s1` nor `s2` shall be a null pointer. Neither `s1max` nor `n` shall be greater than `RSIZE_MAX/sizeof(wchar_t)`. `n` shall not be greater than `s1max`.

3 If there is a runtime-constraint violation, the `wmemmove_s` function stores zeros in the first `s1max` wide characters of the object pointed to by `s1` if `s1` is not a null pointer and `s1max` is not greater than `RSIZE_MAX/sizeof(wchar_t)`.

Description

4 The `wmemmove_s` function copies `n` successive wide characters from the object pointed to by `s2` into the object pointed to by `s1`. This copying takes place as if the `n` wide characters from the object
pointed to by \texttt{s2} are first copied into a temporary array of \textit{n} wide characters that does not overlap the objects pointed to by \texttt{s1} or \texttt{s2}, and then the \textit{n} wide characters from the temporary array are copied into the object pointed to by \texttt{s1}.

**Returns**

The \texttt{wmemmove_s} function returns zero if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

K.3.9.2.2 Wide string concatenation functions


\subsection{The \texttt{wcscat_s} function}

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
#endif

errno_t wcscat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2);
```

**Runtime-constraints**

2. Let \( m \) denote the value \( \text{slmax} - \text{wcsnlen_s}(\text{s1, slmax}) \) upon entry to \texttt{wcscat_s}.

3. Neither \texttt{s1} nor \texttt{s2} shall be a null pointer. \texttt{slmax} shall not be greater than \( \text{RSIZE_MAX} / \text{sizeof(wchar_t)} \). \texttt{slmax} shall not equal zero. \( m \) shall not equal zero.\(^{473} \) \( m \) shall not equal zero.\(^{474} \) Copying shall not take place between objects that overlap.

4. If there is a runtime-constraint violation, then if \( \text{s1} \) is not a null pointer and \( \text{slmax} \) is greater than zero and not greater than \( \text{RSIZE_MAX} / \text{sizeof(wchar_t)} \), then \texttt{wcscat_s} sets \texttt{s1}[0] to the null wide character.

**Description**

5. The \texttt{wcscat_s} function appends a copy of the wide string pointed to by \texttt{s2} (including the terminating null wide character) to the end of the wide string pointed to by \texttt{s1}. The initial wide character from \texttt{s2} overwrites the null wide character at the end of \texttt{s1}.

6. All elements following the terminating null wide character (if any) written by \texttt{wcscat_s} in the array of \texttt{slmax} wide characters pointed to by \texttt{s1} take unspecified values when \texttt{wcscat_s} returns.\(^{475} \)

**Returns**

7. The \texttt{wcscat_s} function returns zero\(^{475} \) if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

\subsection{The \texttt{wcsncat_s} function}

**Synopsis**

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
#endif

errno_t wcsncat_s(wchar_t * restrict s1, rsize_t s1max, const wchar_t * restrict s2, rsize_t n);
```

**Runtime-constraints**

2. Let \( m \) denote the value \( \text{slmax} - \text{wcsnlen_s}(\text{s1, slmax}) \) upon entry to \texttt{wcscat_s}.

3. Neither \texttt{s1} nor \texttt{s2} shall be a null pointer. Neither \texttt{slmax} nor \texttt{n} shall be greater than \( \text{RSIZE_MAX} / \text{sizeof(wchar_t)} \). \texttt{slmax} shall not equal zero. \( m \) shall not equal zero.\(^{476} \) If \( n \)

\(^{473} \) Zero means that \texttt{s1} was not null terminated upon entry to \texttt{wcscat_s}.

\(^{474} \) This allows an implementation to append wide characters from \texttt{s2} to \texttt{s1} while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of \texttt{s1} before discovering that the first element was set to the null wide character.

\(^{475} \) A zero return value implies that all of the requested wide characters from the wide string pointed to by \texttt{s2} were appended to the wide string pointed to by \texttt{s1} and that the result in \texttt{s1} is null terminated.

\(^{476} \) Zero means that \texttt{s1} was not null terminated upon entry to \texttt{wcscat_s}.
is not less than \( m \), then \( m \) shall be greater than \( \text{wcsnlen}_s(s2, m) \). Copying shall not take place between objects that overlap.

4 If there is a runtime-constraint violation, then if \( s1 \) is not a null pointer and \( s1max \) is greater than zero and not greater than \( \text{RSIZE\_MAX}/\text{sizeof}(\text{wchar\_t}) \), then \( \text{wcsncat}_s \) sets \( s1[0] \) to the null wide character.

**Description**

The \( \text{wcsncat}_s \) function appends not more than \( n \) successive wide characters (wide characters that follow a null wide character are not copied) from the array pointed to by \( s2 \) to the end of the wide string pointed to by \( s1 \). The initial wide character from \( s2 \) overwrites the null wide character at the end of \( s1 \). If no null wide character was copied from \( s2 \), then \( s1[s1max-m+n] \) is set to a null wide character.

6 All elements following the terminating null wide character (if any) written by \( \text{wcsncat}_s \) in the array of \( s1max \) wide characters pointed to by \( s1 \) take unspecified values when \( \text{wcsncat}_s \) returns.\(^{477}\)

**Returns**

7 The \( \text{wcsncat}_s \) function returns zero\(^{478}\) if there was no runtime-constraint violation. Otherwise, a nonzero value is returned.

**EXAMPLE 1** The \( \text{wcsncat}_s \) function can be used to copy a wide string without the danger that the result will not be null terminated or that wide characters will be written past the end of the destination array.

\[
\begin{align*}
\text{#define __STDC_WANT_LIB_EXT1__} & 1 \\
\text{#include}<\text{wchar.h}> & \\
/* ... */ & \\
\text{wchar\_t } s1[100] = L"good"; & \\
\text{wchar\_t } s2[6] = L"hello"; & \\
\text{wchar\_t } s3[6] = L"hello"; & \\
\text{wchar\_t } s4[7] = L"abc"; & \\
\text{wchar\_t } s5[1000] = L"bye"; & \\
\text{int } r1, r2, r3, r4; & \\
r1 = \text{wcsncat}_s(s1, 100, s5, 1000); & \\
r2 = \text{wcsncat}_s(s2, 6, L"", 1); & \\
r3 = \text{wcsncat}_s(s3, 6, L"X", 2); & \\
r4 = \text{wcsncat}_s(s4, 7, L"defghijklmn", 3);
\end{align*}
\]

After the first call \( r1 \) will have the value zero and \( s1 \) will be the wide character sequence \( \text{goodbye}\0 \). After the second call \( r2 \) will have the value zero and \( s2 \) will be the wide character sequence \( \text{hello}\0 \). After the third call \( r3 \) will have a nonzero value and \( s3 \) will be the wide character sequence \( \0 \). After the fourth call \( r4 \) will have the value zero and \( s4 \) will be the wide character sequence \( \text{abcdef}\0 \).

**K.3.9.2.3** Wide string search functions

**K.3.9.2.3.1** The \( \text{wcstok}_s \) function

**Synopsis**

\[
\begin{align*}
\text{#define __STDC_WANT_LIB_EXT1__} & 1 \\
\text{#include}<\text{wchar.h}> & \\
\text{wchar\_t } *\text{wcstok}_s(\text{wchar\_t } * \text{restrict } s1, \text{size\_t } * \text{restrict } s1max, & \\
\text{const wchar\_t } * \text{restrict } s2, \text{wchar\_t } ** \text{restrict } ptr); & \\
\end{align*}
\]

**Runtime-constraints**

2 None of \( s1max, s2, \) or \( ptr \) shall be a null pointer. If \( s1 \) is a null pointer, then \( *ptr \) shall not be a null pointer. The value of \( *s1max \) shall not be greater than \( \text{RSIZE\_MAX}/\text{sizeof}(\text{wchar\_t}) \). The end of

\(^{477}\)This allows an implementation to append wide characters from \( s2 \) to \( s1 \) while simultaneously checking if any of those wide characters are null. Such an approach might write a wide character to every element of \( s1 \) before discovering that the first element was set to the null wide character.

\(^{478}\)A zero return value implies that all of the requested wide characters from the wide string pointed to by \( s2 \) were appended to the wide string pointed to by \( s1 \) and that the result in \( s1 \) is null terminated.
the token found shall occur within the first \*s1max wide characters of s1 for the first call, and shall occur within the first \*s1max wide characters of where searching resumes on subsequent calls.

If there is a runtime-constraint violation, the \texttt{wcstok\_s} function does not indirect through the s1 or s2 pointers, and does not store a value in the object pointed to by ptr.

\textbf{Description}

A sequence of calls to the \texttt{wcstok\_s} function breaks the wide string pointed to by s1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by s2. The fourth argument points to a caller-provided wchar\_t pointer into which the \texttt{wcstok\_s} function stores information necessary for it to continue scanning the same wide string.

The first call in a sequence has a non-null first argument and s1max points to an object whose value is the number of elements in the wide character array pointed to by the first argument. The first call stores an initial value in the object pointed to by ptr and updates the value pointed to by s1max to reflect the number of elements that remain in relation to ptr. Subsequent calls in the sequence have a null first argument and the objects pointed to by s1max and ptr are required to have the values stored by the previous call in the sequence, which are then updated. The separator wide string pointed to by s2 may be different from call to call.

The first call in the sequence searches the wide string pointed to by s1 for the first wide character that is not contained in the current separator wide string pointed to by s2. If no such wide character is found, then there are no tokens in the wide string pointed to by s1 and the \texttt{wcstok\_s} function returns a null pointer. If such a wide character is found, it is the start of the first token.

The \texttt{wcstok\_s} function then searches from there for the first wide character in s1 that is contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide string pointed to by s1, and subsequent searches in the same wide string for a token return a null pointer. If such a wide character is found, it is overwritten by a null wide character, which terminates the current token.

In all cases, the \texttt{wcstok\_s} function stores sufficient information in the pointer pointed to by ptr so that subsequent calls, with a null pointer for s1 and the unmodified pointer value for ptr, shall start searching just past the element overwritten by a null wide character (if any).

\textbf{Returns}

The \texttt{wcstok\_s} function returns a pointer to the first wide character of a token, or a null pointer if there is no token or there is a runtime-constraint violation.

\textbf{EXAMPLE}

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
static wchar_t str1[] = L"?a??b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t, *ptr1, *ptr2;
rsize_t max1 = wcslen(str1)+1;
rsize_t max2 = wcslen(str2)+1;

t = wcstok_s(str1, &max1, ",", &ptr1); // t points to the token "a"
t = wcstok_s(NULL, &max1, ",", &ptr1); // t points to the token "??b"
t = wcstok_s(str2, &max2, ",\t", &ptr2); // t is a null pointer

t = wcstok_s(NULL, &max1, ",#", &ptr1); // t points to the token "c"
t = wcstok_s(NULL, &max1, ",?", &ptr1); // t is a null pointer
```

\section*{K.3.9.2.4 Miscellaneous functions}

\subsection*{K.3.9.2.4.1 The wcsnlen\_s function}

\textbf{Synopsis}

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include <wchar.h>
size_t wcsnlen_s(const wchar_t *s, size_t maxsize);
```
Description
The **wcsnlen_s** function computes the length of the wide string pointed to by **s**.

Returns
3  If **s** is a null pointer,\(^{479}\) then the **wcsnlen_s** function returns zero.
4  Otherwise, the **wcsnlen_s** function returns the number of wide characters that precede the terminating null wide character. If there is no null wide character in the first **maxsize** wide characters of **s** then **wcsnlen_s** returns **maxsize**. At most the first **maxsize** wide characters of **s** shall be accessed by **wcsnlen_s**.

K.3.9.3  Extended multibyte/wide character conversion utilities
K.3.9.3.1  Restartable multibyte/wide character conversion functions
1  Unlike **wcrtomb**, **wcrtomb_s** does not permit the **ps** parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.1.1  The **wcrtomb_s** function
Synopsis
1  ```
#include <wchar.h>
errno_t wcrtomb_s( size_t * restrict retval, char * restrict s, rsize_t smax, wchar_t wc, mbstate_t * restrict ps);
```  

Runtime-constraints
2  Neither **retval** nor **ps** shall be a null pointer. If **s** is not a null pointer, then **smax** shall not equal zero and shall not be greater than **RSIZE_MAX**. If **s** is not a null pointer, then **smax** shall be not be less than the number of bytes to be stored in the array pointed to by **s**. If **s** is a null pointer, then **smax** shall equal zero.
3  If there is a runtime-constraint violation, then **wcrtomb_s** does the following. If **s** is not a null pointer and **smax** is greater than zero and not greater than **RSIZE_MAX**, then **wcrtomb_s** sets **s[0]** to the null character. If **retval** is not a null pointer, then **wcrtomb_s** sets **retval** to (size_t)\((-1)\).

Description
4  If **s** is a null pointer, the **wcrtomb_s** function is equivalent to the call

```
wcrtomb_s(&retval, buf, sizeof buf, L'\0', ps)
```  

where **retval** and **buf** are internal variables of the appropriate types, and the size of **buf** is greater than **MB_CUR_MAX**.
5  If **s** is not a null pointer, the **wcrtomb_s** function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by **wc** (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by **s**. At most **MB_CUR_MAX** bytes are stored. If **wc** is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.
6  If **wc** does not correspond to a valid multibyte character, an encoding error occurs: the **wcrtomb_s** function stores the value (size_t)(-1) into **retval** and the conversion state is unspecified. Otherwise, the **wcrtomb_s** function stores into **retval** the number of bytes (including any shift sequences) stored in the array pointed to by **s**.

\(^{479}\)Note that the **wcsnlen_s** function has no runtime-constraints. This lack of runtime-constraints along with the values returned for a null pointer or an unterminated wide string argument make **wcsnlen_s** useful in algorithms that gracefully handle such exceptional data.
Returns

7 The \texttt{wcrtomb\_s} function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2 Restartable multibyte/wide string conversion functions

1 Unlike \texttt{mbsrtowcs} and \texttt{wcsrtombs}, \texttt{mbsrtowcs\_s} and \texttt{wcsrtombs\_s} do not permit the \texttt{ps} parameter (the pointer to the conversion state) to be a null pointer.

K.3.9.3.2.1 The \texttt{mbsrtowcs\_s} function

Synopsis

1

\begin{verbatim}
#include <wchar.h>
errno_t mbsrtowcs_s(size_t * restrict retval, wchar_t * restrict dst, rsize_t dstmax, const char ** restrict src, rsize_t len, mbstate_t * restrict ps);
\end{verbatim}

Runtime-constraints

2 None of \texttt{retval}, \texttt{src}, or \texttt{ps} shall be null pointers. If \texttt{dst} is not a null pointer, then neither \texttt{len} nor \texttt{dstmax} shall be greater than \texttt{RSIZE\_MAX}/\texttt{sizeof(wchar\_t)}. If \texttt{dst} is a null pointer, then \texttt{dstmax} shall equal zero. If \texttt{dst} is not a null pointer, then \texttt{dstmax} shall not equal zero. If \texttt{dst} is not a null pointer and \texttt{len} is not less than \texttt{dstmax}, then a null character shall occur within the first \texttt{dstmax} multibyte characters of the array pointed to by \texttt{*src}.

3 If there is a runtime-constraint violation, then \texttt{mbsrtowcs\_s} does the following. If \texttt{retval} is not a null pointer, then \texttt{mbsrtowcs\_s} sets \texttt{*retval} to \texttt{(-1)}. If \texttt{dst} is not a null pointer and \texttt{dstmax} is greater than zero and not greater than \texttt{RSIZE\_MAX}/\texttt{sizeof(wchar\_t)}, then \texttt{mbsrtowcs\_s} sets \texttt{dst[0]} to the null wide character.

Description

4 The \texttt{mbsrtowcs\_s} function converts a sequence of multibyte characters that begins in the conversion state described by the object pointed to by \texttt{ps}, from the array indirectly pointed to by \texttt{src} into a sequence of corresponding wide characters. If \texttt{dst} is not a null pointer, the converted characters are stored into the array pointed to by \texttt{dst}. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier in two cases: when a sequence of bytes is encountered that does not form a valid multibyte character, or (if \texttt{dst} is not a null pointer) when \texttt{len} wide characters have been stored into the array pointed to by \texttt{*src}.

5 If \texttt{dst} is not a null pointer, the pointer object pointed to by \texttt{src} is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last multibyte character converted (if any). If conversion stopped due to reaching a terminating null character and if \texttt{dst} is not a null pointer, the resulting state described is the initial conversion state.

6 Regardless of whether \texttt{dst} is or is not a null pointer, if the input conversion encounters a sequence of bytes that do not form a valid multibyte character, an encoding error occurs: the \texttt{mbsrtowcs\_s} function stores the value \texttt{(size\_t)(-1)} into \texttt{*retval} and the conversion state is unspecified. Otherwise, the \texttt{mbsrtowcs\_s} function stores into \texttt{*retval} the number of multibyte characters successfully converted, not including the terminating null character (if any).

7 All elements following the terminating null wide character (if any) written by \texttt{mbsrtowcs\_s} in the array of \texttt{dstmax} wide characters pointed to by \texttt{dst} take unspecified values when \texttt{mbsrtowcs\_s} returns.

8 If copying takes place between objects that overlap, the objects take on unspecified values.

\footnote{480}This, the value of \texttt{len} is ignored if \texttt{dst} is a null pointer.

\footnote{481}This allows an implementation to attempt converting the multibyte string before discovering a terminating null character did not occur where required.
Returns

9 The \texttt{mbstowcs\_s} function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.

K.3.9.3.2.2 The \texttt{wcsrtombs\_s} function

Synopsis

1

```c
#include <wchar.h>
errno_t wcsrtombs_s(size_t * restrict retval, char * restrict dst,
                    size_t dstmax, const wchar_t ** restrict src, size_t len,
                    mbstate_t * restrict ps);
```

Runtime-constraints

2 None of \texttt{retval}, \texttt{src}, \texttt{*src}, or \texttt{ps} shall be null pointers. If \texttt{dst} is not a null pointer, then neither \texttt{len} shall be greater than \texttt{RSIZE\_MAX/sizeof(wchar\_t)} nor \texttt{dstmax} shall be greater than \texttt{RSIZE\_MAX}. If \texttt{dst} is a null pointer, then \texttt{dstmax} shall equal zero. If \texttt{dst} is not a null pointer, then \texttt{dstmax} shall not equal zero. If \texttt{dst} is not a null pointer and \texttt{len} is not less than \texttt{dstmax}, then the conversion shall have been stopped (see below) because a terminating null wide character was reached or because an encoding error occurred.

3 If there is a runtime-constraint violation, then \texttt{wcsrtombs\_s} does the following. If \texttt{retval} is not a null pointer, then \texttt{wcsrtombs\_s} sets \texttt{retval} to \texttt{(size_t)(-1)}. If \texttt{dst} is not a null pointer and \texttt{dstmax} is greater than zero and not greater than \texttt{RSIZE\_MAX}, then \texttt{wcsrtombs\_s} sets \texttt{dst[0]} to the null character.

Description

4 The \texttt{wcsrtombs\_s} function converts a sequence of wide characters from the array indirectly pointed to by \texttt{src} into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by \texttt{ps}. If \texttt{dst} is not a null pointer, the converted characters are then stored into the array pointed to by \texttt{dst}. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier in two cases:

— when a wide character is reached that does not correspond to a valid multibyte character;

— (if \texttt{dst} is not a null pointer) when the next multibyte character would exceed the limit of \texttt{n} total bytes to be stored into the array pointed to by \texttt{dst}. If the wide character being converted is the null wide character, then \texttt{n} is the lesser of \texttt{len} or \texttt{dstmax}. Otherwise, \texttt{n} is the lesser of \texttt{len} or \texttt{dstmax-1}.

If the conversion stops without converting a null wide character and \texttt{dst} is not a null pointer, then a null character is stored into the array pointed to by \texttt{dst} immediately following any multibyte characters already stored. Each conversion takes place as if by a call to the \texttt{wcrtomb} function.\footnote{If conversion stops because a terminating null wide character has been reached, the bytes stored include those necessary to reach the initial shift state immediately before the null byte. However, if the conversion stops before a terminating null wide character has been reached, the result will be null terminated, but might not end in the initial shift state.}

5 If \texttt{dst} is not a null pointer, the pointer object pointed to by \texttt{src} is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

6 Regardless of whether \texttt{dst} is or is not a null pointer, if the input conversion encounters a wide character that does not correspond to a valid multibyte character, an encoding error occurs: the \texttt{wcsrtombs\_s} function stores the value \texttt{(size_t)(-1)} into \texttt{*retval} and the conversion state is unspecified. Otherwise, the \texttt{wcsrtombs\_s} function stores into \texttt{*retval} the number of bytes in the resulting multibyte character sequence, not including the terminating null character (if any).

7 All elements following the terminating null character (if any) written by \texttt{wcsrtombs\_s} in the array
of \texttt{dstmax} elements pointed to by \texttt{dst} take unspecified values when \texttt{wcsrtombs_s} returns.\footnote{When \texttt{len} is not less than \texttt{dstmax}, the implementation might fill the array before discovering a runtime-constraint violation.}

8 If copying takes place between objects that overlap, the objects take on unspecified values.

\textbf{Returns}

9 The \texttt{wcsrtombs_s} function returns zero if no runtime-constraint violation and no encoding error occurred. Otherwise, a nonzero value is returned.
Annex L
(normative)

Analyzability

L.1 Scope
1 This annex specifies optional behavior that can aid in the analyzability of C programs.
2 An implementation that defines __STDC_ANALYZABLE__ shall conform to the specifications in this annex.

L.2 Definitions
L.2.1 out-of-bounds store
an (attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for
an object declared volatile, fetch) one or more bytes that lie outside the bounds permitted by this
document.

L.2.2 bounded undefined behavior
undefined behavior (3.4.3) that does not perform an out-of-bounds store.
1 Note 1 to entry: The behavior might perform a trap.
2 Note 2 to entry: Any values produced or stored might be indeterminate values.

L.2.3 critical undefined behavior
undefined behavior that is not bounded undefined behavior.
1 Note 1 to entry: The behavior might perform an out-of-bounds store or perform a trap.

L.3 Requirements
1 If the program performs a trap (3.19.5), the implementation is permitted to invoke a runtime-
constraint handler. Any such semantics are implementation-defined.
2 All undefined behavior shall be limited to bounded undefined behavior, except for the following
which are permitted to result in critical undefined behavior:
   — An object is referred to outside of its lifetime (6.2.4).
   — A store is performed to an object that has two incompatible declarations (6.2.7),
   — A pointer is used to call a function whose type is not compatible with the referenced type
      (6.2.7, 6.3.2.3, 6.5.2.2).
   — An lvalue does not designate an object when evaluated (6.3.2.1).
   — The program attempts to modify a string literal (6.4.5).
   — The operand of the unary * operator has an invalid value (6.5.3.2).
   — Addition or subtraction of a pointer into, or just beyond, an array object and an integer type
      produces a result that points just beyond the array object and is used as the operand of a unary
      * operator that is evaluated (6.5.6).
   — An attempt is made to modify an object defined with a const-qualified type through use of an
      lvalue with non-const-qualified type (6.7.3).

484) Implementations that do not define __STDC_ANALYZABLE__ are not required to conform to these specifications.
— An argument to a function or macro defined in the standard library has an invalid value or a type not expected by a function with variable number of arguments (7.1.4).

— The `longjmp` function is called with a `jmp_buf` argument where the most recent invocation of the `setjmp` macro in the same invocation of the program with the corresponding `jmp_buf` argument is nonexistent, or the invocation was from another thread of execution, or the function containing the invocation has terminated execution in the interim, or the invocation was within the scope of an identifier with variably modified type and execution has left that scope in the interim (7.13.2.1).

— The value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is used (7.22.3).

— A string or wide string utility function accesses an array beyond the end of an object (7.24.1, 7.29.4).
Annex M
(informative)
Change History

M.1 Fifth Edition
Major changes in this fifth edition (\__STDC_VERSION\__ yyyymmL) include:

— added a one-argument version of _Static_assert
— support for function definitions with identifier lists has been removed
— harmonization with ISO/IEC 9945 (POSIX):
  • extended month name formats for strftime
  • integration of functions: asctime_r, ctime_r, gmtime_r, localtime_r, memccpy, strdup, strndup
— harmonization with floating point standard IEC 60559:
  • integration of binary floating-point technical specification TS 18661-1
  • integration of decimal floating-point technical specification TS 18661-2
  • integration of decimal floating-point technical specification TS 18661-4a
— the macro DECIMAL_DIG is declared obsolescent
— added version test macros to certain library headers
— added the attributes feature
— added deprecated, fallthrough, maybe_unused, and nodiscard attributes
— added the u8 character prefix

M.2 Fourth Edition
There were no major changes in the fourth edition (\__STDC_VERSION\__ 201710L), only technical corrections and clarifications.

M.3 Third Edition
Major changes in the third edition (\__STDC_VERSION\__ 201112L) included:

— conditional (optional) features (including some that were previously mandatory)
— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (<stdatomic.h> and <threads.h>)
— additional floating-point characteristic macros (<float.h>)
— querying and specifying alignment of objects (<stdalign.h>, <stdlib.h>)
— Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 19769:2004)
— type-generic expressions
— static assertions
— anonymous structures and unions
— no-return functions
— macros to create complex numbers (<complex.h>)
— support for opening files for exclusive access
— removed the `gets` function (`<stdio.h>`)  
— added the `aligned_alloc`, `at_quick_exit`, and `quick_exit` functions (`<stdlib.h>`)  
— (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731–1:2007)  
— (conditional) support for analyzability

M.4 Second Edition

1 Major changes in the second edition (`__STDC_VERSION__ 199901L`) included:

— restricted character set support via digraphs and `<iso646.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)  
— wide character library support in `<wchar.h>` and `<wctype.h>` (originally specified in ISO/IEC 9899:1990/Amd 1:1995)  
— more precise aliasing rules via effective type  
— restricted pointers  
— variable length arrays  
— flexible array members  
— `static` and type qualifiers in parameter array declarators  
— complex (and imaginary) support in `<complex.h>`  
— type-generic math macros in `<tgmath.h>`  
— the `long long int` type and library functions  
— extended integer types  
— increased minimum translation limits  
— additional floating-point characteristics in `<float.h>`  
— remove implicit `int`  
— reliable integer division  
— universal character names (`\u` and `\U`)  
— extended identifiers  
— hexadecimal floating-point constants and `%a` and `%A` `printf/scanf` conversion specifiers  
— compound literals  
— designated initializers  
— `//` comments  
— specified width integer types and corresponding library functions in `<inttypes.h>` and `<stdint.h>`  
— remove implicit function declaration  
— preprocessor arithmetic done in `intmax_t/uintmax_t`  
— mixed declarations and statements

§ M.4 Change History
new block scopes for selection and iteration statements
integer constant type rules
integer promotion rules
macros with a variable number of arguments (\texttt{\_VAARGS\_})
the \texttt{vscanf} family of functions in \texttt{<stdio.h>} and \texttt{<wchar.h>}
additional math library functions in \texttt{<math.h>}
treatment of error conditions by math library functions (\texttt{math_errhandling})
floating-point environment access in \texttt{<fenv.h>}
IEC 60559 (also known as IEC 559 or IEEE arithmetic) support
trailing comma allowed in \texttt{enum} declaration
\texttt{\%lf} conversion specifier allowed in \texttt{printf}
inline functions
the \texttt{snprintf} family of functions in \texttt{<stdio.h>}
boolean type in \texttt{<stdbool.h>}
idempotent type qualifiers
empty macro arguments
new structure type compatibility rules (tag compatibility)
additional predefined macro names
\texttt{\_Pragma} preprocessing operator
standard pragmas
\texttt{\_func\_} predefined identifier
\texttt{\_va\_copy} macro
additional \texttt{strftime} conversion specifiers
LIA compatibility annex
deprecate \texttt{ungetc} at the beginning of a binary file
remove deprecation of aliased array parameters
conversion of array to pointer not limited to lvalues
relaxed constraints on aggregate and union initialization
relaxed restrictions on portable header names
\texttt{return} without expression not permitted in function that returns a value (and vice versa)

\textbf{M.5 First Edition, Amendment 1}

Major changes in the amendment to the first edition (\texttt{\_STDVERSION\_ 199409L}) included:

- addition of the predefined \texttt{\_STDVERSION\_} macro
- restricted character set support via digraphs and \texttt{<iso646.h>}
- wide character library support in \texttt{<wchar.h>} and \texttt{<wctype.h>}

Bibliography


# ISO/IEC 9899:202x (E)  working draft — November 18, 2019  N2455

## Index

<table>
<thead>
<tr>
<th>Operator</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>! (exclamation-mark punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>! (logical negation operator)</td>
<td>73</td>
</tr>
<tr>
<td>!= (inequality operator)</td>
<td>79</td>
</tr>
<tr>
<td>!= (not-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td># (hash punctuator)</td>
<td>61</td>
</tr>
<tr>
<td># preprocess directive</td>
<td>147</td>
</tr>
<tr>
<td># punctuator, 135</td>
<td></td>
</tr>
<tr>
<td>## (hash-hash punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>#define preprocess directive</td>
<td>139</td>
</tr>
<tr>
<td>#elif preprocess directive</td>
<td>137</td>
</tr>
<tr>
<td>#else preprocess directive</td>
<td>137</td>
</tr>
<tr>
<td>#endif preprocess directive</td>
<td>137</td>
</tr>
<tr>
<td>#error preprocess directive</td>
<td>8, 146</td>
</tr>
<tr>
<td>#if preprocess directive</td>
<td>21, 24, 137, 154</td>
</tr>
<tr>
<td>#ifdef preprocess directive</td>
<td>137</td>
</tr>
<tr>
<td>#ifndef preprocess directive</td>
<td>8, 152</td>
</tr>
<tr>
<td>#ifdef preprocess directive</td>
<td>8, 152</td>
</tr>
<tr>
<td>#include preprocess directive</td>
<td>9, 138</td>
</tr>
<tr>
<td>#line preprocess directive</td>
<td>145</td>
</tr>
<tr>
<td>#pragma preprocess directive</td>
<td>146</td>
</tr>
<tr>
<td>#undef preprocess directive</td>
<td>142, 154</td>
</tr>
<tr>
<td>$[x]$, 7</td>
<td></td>
</tr>
<tr>
<td>$[x]$, 7</td>
<td></td>
</tr>
<tr>
<td>&amp; (bitwise AND operator)</td>
<td>40, 79</td>
</tr>
<tr>
<td>&amp;&amp; (logical AND operator)</td>
<td>15, 80</td>
</tr>
<tr>
<td>&amp;= (bitwise AND assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>' ' (space character)</td>
<td>9, 17, 48, 166, 168, 387</td>
</tr>
<tr>
<td>( opening parenthesis punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>) (cast operator)</td>
<td>75</td>
</tr>
<tr>
<td>) (function-call operator)</td>
<td>67</td>
</tr>
<tr>
<td>) (parentheses punctuator)</td>
<td>108, 125, 127</td>
</tr>
<tr>
<td>) {} (compound-literal operator)</td>
<td>71</td>
</tr>
<tr>
<td>) (closing parenthesis punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>* (asterisk punctuator)</td>
<td>61, 105, 106</td>
</tr>
<tr>
<td>* (indirection operator)</td>
<td>67, 73</td>
</tr>
<tr>
<td>* (multiplication operator)</td>
<td>40, 75, 472</td>
</tr>
<tr>
<td>*= (asterisk-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>+= (multiplication assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>+ (addition operator)</td>
<td>40, 67, 73, 76, 474</td>
</tr>
<tr>
<td>+ (plus punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>+ (unary plus operator)</td>
<td>73</td>
</tr>
<tr>
<td>++ (plus-plus punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>++ (postfix increment operator)</td>
<td>46, 70</td>
</tr>
<tr>
<td>++ (prefix increment operator)</td>
<td>46, 73</td>
</tr>
<tr>
<td>+= (addition assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>+= (plus-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>, (comma operator)</td>
<td>15, 84</td>
</tr>
<tr>
<td>, (comma punctuator)</td>
<td>61, 66, 87, 91, 94, 96, 112</td>
</tr>
<tr>
<td>- (minus punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>- (subtraction operator)</td>
<td>40, 76, 474</td>
</tr>
<tr>
<td>- (unary minus operator)</td>
<td>73, 444</td>
</tr>
<tr>
<td>- format flag, 276, 354</td>
<td></td>
</tr>
<tr>
<td>- (minus-minus punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>- (postfix decrement operator)</td>
<td>46, 70</td>
</tr>
<tr>
<td>- (prefix decrement operator)</td>
<td>46, 73</td>
</tr>
<tr>
<td>-= (minus-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>-&gt; (subtraction assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>-&gt; (minus-greater punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>-&gt; (structure/union pointer operator)</td>
<td>69</td>
</tr>
<tr>
<td>. (dot punctuator)</td>
<td>61, 112</td>
</tr>
<tr>
<td>. (structure/union member operator)</td>
<td>46, 68</td>
</tr>
<tr>
<td>. . . (ellipsis punctuator)</td>
<td>61, 68, 108, 139</td>
</tr>
<tr>
<td>/ (division operator)</td>
<td>40, 75, 472</td>
</tr>
<tr>
<td>/ (slash punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>/= (comment delimiters)</td>
<td>62</td>
</tr>
<tr>
<td>// (comment delimiter)</td>
<td>63</td>
</tr>
<tr>
<td>/= (division assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>/= (slash-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>: (colon punctuator)</td>
<td>91</td>
</tr>
<tr>
<td>: (colon greater punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;= (less-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;= (less-than-or-equal-to operator)</td>
<td>78</td>
</tr>
<tr>
<td>%= (alternative spelling of %)</td>
<td>61</td>
</tr>
<tr>
<td>%= (less-percent punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt; (less punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt; (less-than operator)</td>
<td>78</td>
</tr>
<tr>
<td>: (alternative spelling of [])</td>
<td>61</td>
</tr>
<tr>
<td>: (less-colon punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;= (left-shift operator)</td>
<td>40, 77</td>
</tr>
<tr>
<td>&lt;= (less-less punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;&lt;= (left-shift assignment operator)</td>
<td>83</td>
</tr>
<tr>
<td>&lt;&lt;= (less-less equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;&lt;= (less-equal punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;= (less-than-or-equal-to operator)</td>
<td>78</td>
</tr>
<tr>
<td>%= (alternative spelling of %)</td>
<td>61</td>
</tr>
<tr>
<td>%= (less-percent punctuator)</td>
<td>61</td>
</tr>
<tr>
<td>&lt;assert.h&gt; header, 152, 153, 156, 180, 410</td>
<td></td>
</tr>
<tr>
<td>&lt;ctype.h&gt; header, 152, 166, 167, 168, 392, 411</td>
<td></td>
</tr>
<tr>
<td>&lt;errno.h&gt; header, 152, 170, 392, 411, 523</td>
<td></td>
</tr>
<tr>
<td>&lt;fenv.h&gt; header, 8, 12, 14, 24, 29, 32, 84, 152, 171, 173–175, 177–182, 198, 392, 411, 442, 444, 448–450, 459, 463–465, 485, 574</td>
<td></td>
</tr>
<tr>
<td>&lt;float.h&gt; header, 8, 20, 23, 24, 27, 28, 29, 152, 183, 280, 303, 358, 371, 392, 412, 438, 447, 505, 572, 573</td>
<td></td>
</tr>
<tr>
<td>&lt;inttypes.h&gt; header, 152, 184, 185, 186, 392, 413, 573</td>
<td></td>
</tr>
<tr>
<td>&lt;iso646.h&gt; header, 8, 152, 187, 413, 573, 574</td>
<td></td>
</tr>
<tr>
<td>&lt;limits.h&gt; header, 8, 20, 21, 36, 37, 152, 188, 413, 438, 505</td>
<td></td>
</tr>
<tr>
<td>&lt;locale.h&gt; header, 153, 189, 190, 392, 413</td>
<td></td>
</tr>
</tbody>
</table>
<setjmp.h> header, 153, 240, 241, 424
<signal.h> header, 153, 242, 243, 393, 424
<stdalign.h> header, 8, 153, 245, 425, 572
<stdio.h> header, 8, 24, 29, 32, 65, 153, 175, 184, 260, 288, 426, 524
<stdio.h> header, 8, 20, 137, 153, 184, 216, 218, 219, 224, 261, 262, 266, 393, 426, 505, 524, 573
<stdlib.h> header, 8, 24, 29, 32, 153, 155, 175, 176, 300, 301–303, 305–312, 314–317, 393, 428, 502, 523, 537, 538–542, 572, 573
<stdnoreturn.h> header, 8, 153, 318, 429
<string.h> header, 153, 319, 320–326, 393, 429, 543, 544–550
<tgmath.h> header, 32, 153, 327, 331, 430, 440, 453, 481, 573
<threads.h> header, 149, 152, 153, 332, 333–340, 394, 431, 572
<time.h> header, 153, 332, 342, 343–347, 380, 393, 431, 550, 551, 552
<uchar.h> header, 58–60, 153, 350, 351, 352, 432, 572
<wctype.h> header, 153, 386, 387–391, 394, 434, 573, 574

= (equal-sign punctuator), 61, 87, 94, 112
= (simple assignment operator), 83
== (equal-equal punctuator), 61
== (equality operator), 79
> (greater punctuator), 61
> (greater-than operator), 78
>= (greater-equal punctuator), 61
>= (greater-than-or-equal-to operator), 78
>> (greater greater punctuator), 61
>>> (greater-greater-equal punctuator), 61
>>> (right-shift assignment operator), 83
? (question-mark punctuator), 61
?: (conditional operator), 15, 81
?? (trigraph sequences), 18
[ (opening bracket punctuator), 61
[ ] (array subscript operator), 67, 73
[ ] (brackets punctuator), 106, 112
# format flag, 276, 355
% (percent punctuator), 61
% (remainder operator), 40, 75
%: (alternative spelling of #), 61
%: (percent-colon punctuator), 61
%:%: (alternative spelling of %), 61
%: (percent-percent punctuator), 61
%: (alternative spelling of !), 61
%A conversion specifier, 278, 347, 357
%B conversion specifier, 347
%C conversion specifier, 347
%D conversion specifier, 347
%E conversion specifier, 278, 356
%F conversion specifier, 278, 347, 356
%G conversion specifier, 278, 347, 356
%H conversion specifier, 347
%I conversion specifier, 347
%M conversion specifier, 347
%N conversion specifier, 347
%O conversion specifier, 347
%S conversion specifier, 347
%T conversion specifier, 348
%U conversion specifier, 348
%V conversion specifier, 348
%W conversion specifier, 348
%X conversion specifier, 278, 348, 356
%Y conversion specifier, 348
%Z conversion specifier, 348
% (conversion specifier), 285, 361
% (conversion specifier), 280, 286, 358, 362
% (conversion specifier), 278, 285, 347, 357, 361
% (conversion specifier), 278, 285, 347, 357, 361
%b conversion specifier, 347
%c conversion specifier, 279, 285, 347, 357, 361
d conversion specifier, 278, 284, 347, 356, 360
e conversion specifier, 278, 285, 347, 356, 361
%f conversion specifier, 278, 285, 356, 361
g conversion specifier, 278, 285, 347, 356, 361
%h conversion specifier, 347
%i conversion specifier, 278, 356
%j conversion specifier, 347
%k conversion specifier, 347
%l conversion specifier, 278, 356
%o conversion specifier, 278, 280, 356, 361
%p conversion specifier, 280, 286, 347, 358, 362
%r conversion specifier, 347

Index 577
%s conversion specifier, 279, 285, 357, 361
%t conversion specifier, 234
%u conversion specifier, 279, 284, 348, 356, 361
%w conversion specifier, 234
%x conversion specifier, 279, 284, 348, 356, 361
%y conversion specifier, 234
%z conversion specifier, 348
& (address operator), 46, 73
& (ampersand operator), 61
&= (ampersand-equal operator), 61
&b (ampersand-ampersand operator), 61
\ (vertical-tab escape sequence), 58, 60
\ (backslash escaped sequence), 58, 149
\" (double-quote escape sequence), 58, 149
\' (single-quote escape sequence), 58, 149
\ (null character), 17, 59, 60
\p (paddling of binary stream), 269
\? (question-mark escape sequence), 58
\U (universal character name), 51
\a (alert escape sequence), 471
\b (backspace escape sequence), 471
\c (character escape sequence), 471
\e (end-of-line escape sequence), 471
\f (form-feed escape sequence), 471
\n (new-line escape sequence), 471
\o (octal digits), 58
\r (carriage-return escape sequence), 471
\t (horizontal-tab escape sequence), 471
\u (universal character name), 51
\v (vertical-tab escape sequence), 471
Alignas, 41, 49, 103, 104, 245, 395, 403, 508
Alignof operator, 41, 42, 46, 49, 72, 74, 85, 104, 106, 131, 245, 395, 400, 492, 505, 508
Atomic type qualifier, 97
Atomic type specifier, 97
Boolean type conversions, 43
C identifier suffix, 266, 393, 426, 508
Complex type, 89, 157
Complex types, 25, 37, 45, 49, 89, 90, 157, 395, 402, 471, 472, 473, 509
Complex_I macro, 157, 410, 475, 509
_DECIMAL_DIG identifier prefix, 27, 280, 303, 358, 371, 447, 448
__Decimal32_t type, 195, 503, 510
__Decimal64_t type, 195, 503, 510
__EXT__ identifier suffix, 440, 510
Exit function, 243, 310, 311, 428, 496, 505, 510
Generic identifier prefix, 153
_Generic_ macro, 49, 65, 66, 330, 395, 400, 511
_INDEX_ macro, 267, 275, 426, 511
__IOFBF__ macro, 267, 275, 426, 511
__IONBF__ macro, 267, 275, 426, 511
Imaginary type, 157
Imaginary types, 471
Imaginary_I macro, 157, 164, 410, 474, 475, 511
__MAX__ identifier suffix, 44, 263, 264, 393, 413, 426, 508
__MIN__ identifier suffix, 263, 264, 393, 413, 426, 508
__Pragma__ operator, 149
__Static_assert__ xii, 49, 117, 156, 395, 405, 512, 572
__Thread_local__ storage-class specifier, 35, 49, 88, 107, 332, 395, 402, 513
__WIDTH__ identifier suffix, 263, 264, 393, 426, 508, 514
__STDC__ identifier prefix, 151
__STDC_VERSION__ identifier prefix, 153
__STDC_WANT_IEC_60559__ identifier prefix, 440, 512
__DATE__ macro, 148, 503, 509
__FILE__ macro, 147, 148, 156, 510
__LINE__ macro, 145–147, 148, 156, 488, 511
__STDC_ANALYZABLE__ macro, 148, 512, 570
__STDC_HOSTED__ macro, 148, 512
__STDC_IEC_559_COMPLEX__ (obsolete), 23, 149, 151, 471, 512
__STDC_IEC_559_COMPLEX__ macro, 23, 471
__STDC_IEC_559__ (obsolete), 23, 149, 151, 424, 440, 512
__STDC_IEC_559__ macro, 23
__STDC_IEC_60559_BFP__ macro, 8, 23, 149, 424, 440, 467, 468, 512
__STDC_IEC_60559_COMPLEX__ macro, 23, 149, 471, 512
__STDC_IEC_60559_DFP__ macro, 8, 149, 175, 179, 180, 202–231, 232, 233, 234–237, 301, 303, 331, 371, 412, 418, 424, 429, 438–440, 468, 469, 512
__STDC_ISO_10646__ macro, 148, 501, 512
__STDC_LIB_EXT1__ macro, 149, 152, 411, 426, 427, 429, 430, 432, 433, 512, 522
__STDC_MB_MIGHT_NEQ_WC__ macro, 148, 260, 512
__STDC_NO_ATOMICS__ macro, 149, 249, 425, 512
__STDC_NO_COMPLEX__ macro, 149, 157, 410, 512
__STDC_NO_THREADS__ macro, 149, 332, 431, 512
__STDC_NO_VLA__ macro, 149, 512
Index 579

abort function, 103, 156, 242, 243, 250, 270, 309, 428, 496, 505, 514, 538
abort_handler_s function, 429, 514, 538
abs function, 154, 314, 428, 482, 514
abs macro, 154
absolute-value functions complex, 162, 480 integer, 185, 314
real, 216, 459
abstract declarator, 110 abstract machine, 11
access, 98, 570 access (verb), 3 accuracy, see floating-point accuracy acos function, 175, 202, 328, 414, 446, 454, 514 acos type-generic macro, 328 acosd128 function, 176, 202, 418, 514 acosd32 function, 176, 202, 418, 514 acosd64 function, 176, 202, 418, 514 acosf function, 175, 202, 418, 514 acosh function, 207, 328, 331, 414, 447, 456, 514 acosh type-generic macro, 328 acoshd128 function, 207, 419, 514 acoshd32 function, 207, 419, 514 acoshd64 function, 207, 419, 514 acoshf function, 207, 331, 414, 514 acoshl function, 207, 414, 514 acosl function, 175, 202, 414, 514 acospi function, 204, 205, 414, 446, 455, 514 acospi type-generic macro, 329 acospid128 function, 205, 419, 514 acospid32 function, 205, 419, 514 acospid64 function, 205, 419, 514 acospif function, 205, 414, 514 acospl function, 205, 414, 514 acquire fence, 253 acquire operation, 14 active position, 19 add and round to narrower type, 232 addition assignment operator (+=), 83 addition operator (+), 40, 67, 73, 76, 474 additive expressions, 76, 474 address constant, 85 address operator (d), 46, 73 address-free, 254 aggregate initialization, 113 aggregate types, 38 alert, 19 alert escape sequence (\a), 19, 58 aliasing, 64 alignas macro, 245, 425, 514 aligned_alloc function, 307, 308, 428, 489, 498, 505, 514, 573 alignment, 3, 41, 308 pointer, 38, 46 structure/union member, 92
alignment header, 245
alignment specifier, 103
alignof macro, 245, 425, 514
allocated storage, order and contiguity, 307
alternative spellings header, 187
and macro, 145, 187, 413, 514
AND operators
  bitwise (\&), 40
  bitwise (\&) , 79
  bitwise assignment (\&=), 83
logical (\&\&), 80
AND operators
  logical (\&\&), 15
and_eq macro, 187, 413, 514
anonymous structure, 92
anonymous union, 92
ANSI/IEEE 754, 440
ANSI/IEEE 854, 440
argc (main function parameter), 11
argument, 3
  array, 132
    default promotions, 68
    function, 67, 132
  macro, substitution, 140
argument, complex, 163
argv (main function parameter), 11
arithmetic constant expression, 85
arithmetic conversions, usual, see usual arithmetic conversions
arithmetic operators
  additive, 40, 76, 474
  bitwise, 40, 73, 79, 80
  increment and decrement, 70, 72
  multiplicative, 40, 75, 472
  shift, 40, 77
  unary, 73
arithmetic types, 37
arithmetic, pointer, 76
array
  argument, 132
  declarator, 106
  initialization, 113
  multidimensional, 67
  parameter, 132
  storage order, 67
  subscript operator ([ ]), 67, 73
  subscripting, 67
  type, 37
  type conversion, 46
  variable length, 105, 106, 149
arrow operator (\textasciitilde\rightarrow), 69
as-if rule, 12
ASCII code set, 18
asctime function, 148, 345, 346, 432, 500, 514
asctime_r function, xiii, 345, 346, 432, 572
asctime_s function, 432, 514, 550, 551, 552
asin function, 203, 328, 414, 446, 454, 481, 514
asin type-generic macro, 328
asin type-generic macro, 481
asind128 function, 203, 419, 514
asind32 function, 203, 419, 514
asind64 function, 203, 419, 514
asinf function, 203, 414, 514
asinh function, 207, 208, 328, 415, 447, 456, 481, 514
asinh type-generic macro, 328
asinh type-generic macro, 481
asinh_d128 function, 207, 419, 514
asinh_d32 function, 207, 419, 514
asinh_d64 function, 207, 419, 514
asinhf function, 207, 415, 514
asinhl function, 207, 415, 514
asinl function, 203, 414, 514
asinpi function, 205, 414, 446, 455, 514
asinpi type-generic macro, 329
asinpi_d128 function, 205, 419, 514
asinpi_d32 function, 205, 419, 514
asinpi_d64 function, 205, 419, 514
asinpif function, 205, 414, 514
asinpi1 function, 205, 414, 514
assert macro, 120, 156, 180, 410, 495, 503, 514
assignment
  compound, 83
  conversion, 83
  expression, 82
  operators, 46, 82
  simple, 83
associativity of operators, 64
asterisk punctuator (*), 105, 106
at_quick_exit function, 309, 310, 311, 428, 489, 498, 515, 573
atan function, 203, 281, 328, 331, 359, 414, 447, 454, 481, 514
atan type-generic macro, 328
atan type-generic macro, 481
atan2 function, 203, 328, 414, 447, 453, 454, 475, 514
atan2 type-generic macro, 329
atan2_d128 function, 203, 419, 514
atan2_d32 function, 203, 419, 514
atan2_d64 function, 203, 419, 514
atan2f function, 203, 414, 514
atan2l function, 203, 414, 514
atan2pil function, 203, 419, 514
atan2pif function, 203, 419, 514
atan2pid128 function, 203, 419, 514
atan2pid32 function, 203, 419, 514
atan2pid64 function, 203, 419, 514
atan2pil function, 203, 419, 514
atan2pif function, 203, 419, 514
atan2pid128 function, 203, 419, 514
atan2pid32 function, 203, 419, 514
atan2pid64 function, 203, 419, 514
atan2pil function, 203, 419, 514
atomic char16_t function, 208, 328, 415, 447, 456, 481, 514
atomic char32_t function, 514
atomic char function, 208, 328, 415, 447, 456, 481, 514
atomic bool function, 208, 328, 415, 447, 456, 481, 514
atomic fetch_ function, 257, 425, 509
atomic fetch_add function, 509
atomic fetch_add_explicit function, 509
atomic fetch_and function, 509
atomic fetch_and_explicit function, 509
atomic fetch_or function, 509
atomic fetch_or_explicit function, 509
atomic fetch_sub function, 509
atomic fetch_sub_explicit function, 509
atomic fetch_xor function, 509
atomic fetch_xor_explicit function, 509
atomic_flag function, 249, 257, 258, 425, 426, 509
atomic_flag_clear function, 258, 425, 509
atomic_flag_clear_explicit function, 258, 426, 509
atomic_flag_init macro, 249, 257, 258, 425, 509
atomic_flag_test_and_set function, 258, 425, 509
atomic_flag_test_and_set_explicit function, 258, 425, 509
atomic_init function, 250, 425, 509
atomic int type, 250, 254, 425, 509
atomic int_fast16_t type, 255, 425, 509
atomic int_fast32_t type, 255, 425, 509
atomic int_fast64_t type, 255, 425, 509
atomic int_fast8_t type, 255, 425, 509
atomic int_least16_t type, 255, 425, 509
atomic int_least32_t type, 255, 425, 509
atomic int_least64_t type, 255, 425, 509
atomic int_least8_t type, 255, 425, 509
atomic intmax_t type, 249, 425, 509
atomic intptr_t type, 255, 425, 509
atomic is lock free function, 243, 254, 425, 496, 509
atomic long type, 254, 425, 509
atomic llong type, 254, 425, 509
atomic macro, 249, 425, 509
atomic_load function, 255, 257, 425, 509
atomic_load_explicit function, 252, 255, 425, 509
atomic long type, 254, 425, 509
atomic macro, 249, 425, 509
atomic pointer lock free macro, 249, 425, 509
atomic ptrdiff_t type, 255, 425, 509
atomic schar type, 254, 425, 509
atomic short type, 254, 425, 509
atomic macro, 249, 425, 509
atomic_signal_fence function, 253, 254, 425, 509
atomic size_t type, 255, 425, 509
atomic store function, 255, 425, 509
atomic store_explicit function, 252, 255,
atomic_thread_fence function, 253, 425, 509
atomic_uchar type, 254, 425, 509
atomic_uint type, 254, 425, 509
atomic_uint_fast16_t type, 255, 425, 509
atomic_uint_fast32_t type, 255, 425, 509
atomic_uint_fast64_t type, 255, 425, 509
atomic_uint_least16_t type, 255, 425, 509
atomic_uint_least32_t type, 255, 425, 509
atomic_uint_least64_t type, 255, 425, 509
atomic_uint_least8_t type, 255, 425, 509
atomic_uintmax_t type, 255, 425, 509
atomic_uintptr_t type, 255, 425, 509
atomic_uchar type, 254, 425, 509
ATOMIC_VAR_INIT (obsolete), 250, 393, 425, 509
atomic_wchar_t type, 254, 425, 509
ATOMIC_VAR_INIT (lock-free) macro, 249, 425, 509
atomic header, 249, 393
attribute
__deprecated__, 119, 510
__maybe_unused__, 511
__nodiscard__, 118, 512
deprecated, ii, xiv, 88, 93, 118, 119, 120, 121, 516, 572
fallthrough, ii, xiv, 118, 121, 122, 572
maybe_unused, ii, xiv, 118, 120, 519, 572
nodiscard, ii, xiv, 118, 119, 120, 519, 572
attribute declaration, 87
attribute prefixed token, 117
attribute token, 117
attributes, 117
auto storage-class specifier, 49, 88, 107, 127, 131, 395, 402, 515
automatic storage duration, 19, 35
backslash character (\), 9, 17, 57
backslash escape sequence (\), 58, 149
backspace, 19
backspace escape sequence (\b), 19, 58
basic character set, 4, 17
basic types, 37
behavior, 3
binary streams, 268, 295, 297
bit, 4
  high order, 4
  low order, 4
bit-field, 92
bitand macro, 187, 413, 515
bitor macro, 187, 413, 515
bitwise operators, 64
  AND, 40, 79
  AND assignment (&=), 83
complement (~), 40, 73
exclusive OR, 40, 80
exclusive OR assignment (^=), 83
inclusive OR, 40, 80
inclusive OR assignment (|=), 83
shift, 40, 77
blank character, 166
block, 123, 124, 125, 127
block scope, 33
block structure, 33
bold type convention, 33
bool macro, 515
boolean type, 43
boolean type and values header, 259, 393
boolean type conversion, 42, 43
bounded undefined behavior, 570
braces punctuator ({}), 94, 96, 112, 124
brackets operator ([ ]), 67, 73
brackets punctuator ([ ]), 106, 112
branch cuts, 157
broken-down time, 342, 343, 345–347, 551–553
bsearch function, 312, 313, 428, 489, 499, 515
bsearch_s function, 429, 515, 539, 540
btowc function, 357, 359, 381, 433, 515
BUFSIZ macro, 267, 269, 275, 426, 515
byte, 4, 74
byte input/output functions, 268
byte-oriented stream, 269
c identifier prefix, 327–329
C program, 9
c16rtomb function, i, 351, 432, 515
c32rtomb function, 352, 432, 515
cabs function, 162, 328, 411, 475, 515
cabs functions, 475
type-generic macro for, 328
cabsf function, 162, 331, 411, 515
cabsl function, 162, 411, 515
cacos function, 158, 328, 410, 475, 476, 515
cacos functions
type-generic macro for, 328
cacosf function, 158, 410, 515
cacosh function, 160, 328, 410, 476, 515
cacosh functions
type-generic macro for,C 328

cacoshf function, 160, 410, 515
cacosf function, 160, 410, 515
cacospf function, 392, 515
calcdt function, 158, 410, 515
cacospf function, 392, 515
calendar time, 342, 343, 344, 346, 552, 553
call by value, 342

call_once function, 332, 333, 431, 515
calloc function, 307, 308, 428, 489, 498, 505, 515
canonical representation, 24
canonicalize family, 31, 32, 229
canonicalize function, 24, 229, 328, 418, 443, 466, 467, 515
canonicalized128 function, 229, 422, 515
canonicalized32 function, 229, 422, 515
canonicalized64 function, 229, 422, 515
canonicalizedef function, 229, 418, 515
canonicalizedel function, 229, 418, 515
carg function, 163, 329, 331, 411, 475, 481, 515
carg functions, 475
carg type-generic macro, 329
carg type-generic macro, 481
cargf function, 163, 411, 515
cargvl function, 163, 411, 515
carriage return, 19
carriage-return escape sequence (\r), 19, 58, 168
carries a dependency, 15
case label, 123, 125
case mapping functions
character, 168
wide character, 390
extensible, 390
casin function, 159, 328, 410, 475, 515
casin functions, 475
type-generic macro for, 328
casinf function, 159, 410, 515
casinl function, 160, 328, 410, 475, 477, 515
casinh functions

type-generic macro for, 328
casinhf function, 160, 410, 515
casinhl function, 160, 410, 515
casinl function, 159, 410, 515
casinpi function, 392, 515
cast, 75
cast expression, 75
cast operator (()), 75
catan function, 159, 328, 410, 475, 515
catan functions, 475
type-generic macro for, 328
catanf function, 159, 410, 515
catanh function, 160, 161, 328, 410, 475, 477, 515
catanh functions

type-generic macro for, 328
catanhf function, 160, 410, 515
catanhf function, 160, 410, 515
catanh function, 159, 410, 515
catani function, 392, 515
cbrt function, i, 66, 216, 328, 330, 416, 459, 515
cbrt type-generic macro, 329

cbrtd128 function, 216, 421, 515
cbrtd32 function, 216, 420, 515
cbrtd64 function, 216, 420, 515
cbrtf function, 216, 328, 330, 416, 515
cbrtl function, 216, 328, 330, 416, 515
ccompoundn function, 392, 515
ccos function, 159, 328, 410, 475, 515
ccos functions, 475
type-generic macro for, 328
ccosf function, 159, 410, 515
ccosh function, 161, 328, 410, 475, 478, 515
ccosh functions
type-generic macro for, 328
ccoshf function, 161, 410, 515
ccoshl function, 161, 410, 515
ccosl function, 159, 410, 515
ccospi function, 392, 515
cceil function, 197, 221, 328, 331, 416, 442, 462, 463, 464, 484, 515
cceil type-generic macro, 329
ccildd128 function, 32, 221, 421, 515
ccildd32 function, 32, 221, 421, 515
ccildd64 function, 32, 221, 421, 515
cceil function, 221, 416, 515
cceil function, 221, 225, 417, 515
cerf function, 392, 515
cerfc function, 392, 515
cexp function, 161, 162, 328, 410, 479, 480, 515
cexp functions

type-generic macro for, 328
cexp10 function, 392, 515
cexp10m function, 392, 515
cexp2 function, 392, 515
cexp2m function, 392, 515
cexph function, 161, 410, 515
cexp function, 161, 410, 515
cexph function, 392, 515
change history, 572
char type, 89
char type conversion, 43, 44
char16_t type, i, 58, 60, 113, 148, 254, 350, 351, 432, 501, 515
char32_t, 32_t type, i, 58, 60, 113, 148, 254, 350, 351, 352, 432, 501, 515
CHAR_BIT macro, 21, 23, 39, 91, 413, 438, 515
CHAR_MAX macro, 21, 23, 191, 192, 413, 438, 515
CHAR_MIN macro, 21, 23, 37, 413, 438, 515
CHAR_WIDTH macro, 21, 515
character, 4
character array initialization, 113

Index

583
character case mapping functions, 168
  wide character, 390
  extensible, 390
character classification functions, 166
  wide character, 386
  extensible, 389
character constant, 10, 17, 57
character display semantics, 19
character handling header, 166, 190, 392
character input/output functions, 292, 536
  wide character, 366
character sets, 17
character string literal, see string literal
character type conversion, 42
character types, 37, 113
characteristics of floating types header, 183, 392
  cimag function, 163, 164, 329, 331, 411, 473, 475, 481, 515
  cimag functions, 475
  cimag type-generic macro, 329
  cimag type-generic macro, 481
  cimagf function, 163, 411, 515
  cimagl function, 163, 331, 411, 515
cis function, 475
classification functions
  character, 166
  floating-point, 199
  wide character, 386
  extensible, 389
clearerr function, 298, 427, 515
clgamma function, 392, 515
clock function, 342, 343, 431, 505, 515
clock_t function, 342, 343, 431, 505, 515
  CLOCK_PERIOD_SEC macro, 342, 343, 431, 515
clog function, 162, 328, 410, 480, 515
clog functions
  type-generic macro for, 328
clog10 function, 392, 515
clog10p1 function, 392, 515
clog1p function, 392, 515
clog2 function, 392, 515
clog2p1 function, 392, 515
clogf function, 162, 331, 410, 515
clogl function, 162, 411, 515
clogpl function, 392, 515
closing, 270
  CMPLX macro, 157, 163, 164, 411, 515
  CMPLXF macro, 163, 164, 411, 515
  CMPLXL macro, 163, 164, 411, 515
cnd_identifier prefix, 394, 508
cnd_broadcast function, 333, 334, 335, 431, 509
  cnd_destroy function, 334, 431, 509
  cnd_init function, 334, 431, 509
  cnd_signal function, 334, 335, 431, 509
  cnd_t type, 332, 333–335, 431, 509
  cnd_timedwait function, 334, 335, 431, 509
  cnd_wait function, 334, 335, 431, 509
collating sequences, 17
colon punctuator (:) 91
comma operator (,), 15, 84
comma punctuator (,), 66, 87, 91, 94, 96, 112
command processor, 312
comment delimiters (** */ and */, 63
comments, 9, 48, 62
common definitions header, 260
common extensions, 506
common initial sequence, 69
common real type, 44
common warnings, 10, 486
comparison functions, 312, 314, 539, 540
  string, 321
  wide string, 375
  comparison macros, 237
  comparison pointer, 78
compatible type, 40, 90, 98, 105
compl macro, 187, 413, 515
complement operator (~), 40, 73
complete, 36
complex type, 36
complex arithmetic header, 157, 392
complex macro, 114, 157–164, 331, 410, 411, 472, 473, 489, 515
complex numbers, 37, 471
complex type conversion, 44
complex type domain, 37
complex types, 37, 89, 149, 471
compliance, see conformance
  components of time, 342, 550
  composite type, 41
  compound assignment, 83
  compound literal, 71
  compound literals, 70
  compound statement, 124
  compound-literal operator (( ) {}), 71
compound function, 216, 217, 416, 446, 459, 515
compound type, 329
compoundn function, 217, 421, 515
compoundn128 function, 217, 421, 515
compoundn32 function, 216, 421, 515
compoundn64 function, 217, 421, 515
compoundn macro, 216, 416, 515
concatenation functions
  string, 320, 546
  wide string, 375, 563
concatenation, preprocessing, see preprocessing
  concatenation
conceptual models, 9
conditional features, 8, 37, 38, 106, 148, 152, 440, 471, 522, 570
conditional inclusion, 136
conditional operator (?:), 15, 81
conflict, 14
conformance, 8
conforming freestanding implementation, 8
conforming hosted implementation, 8
conforming implementation, 8
conforming program, 8
conj function, 164, 329, 411, 475–480, 515
conj functions, 475
conj type-generic macro, 329
conjf function, 164, 411, 515
conjl function, 164, 411, 515
const type qualifier, 97
const-qualified type, 38, 46, 98
constant expression, 85, 449
constants, 52
as primary expression, 65
character, 57
enumeration, 33, 56
floating, 54
hexadecimal, 52
integer, 52
octal, 52
constraint, 5, 8
constraint_handler_t type, 429, 515, 537
consume operation, 14
content of structure/union/enumeration, 96
contiguity of allocated storage, 307
continue, 49, 128, 129, 395, 406, 515
contracted, 65
contracted expression, 65, 199, 448
control character, 17, 166
control wide character, 386
conversion, 42
arithmetic operands, 42
array argument, 132
array parameter, 132
arrays, 45, 46
boolean, 43
boolean, characters, and integers, 42
by assignment, 83
by return statement, 130
complex types, 44
explicit, 42
function, 46
function argument, 68, 132
function designators, 45
function parameter, 132
imaginary, 471
imaginary and complex, 471
implicit, 42
lvalues, 45
pointer, 46
real and complex, 44
real and imaginary, 471
real floating and integer, 44, 447
real floating types, 44
signed and unsigned integers, 43
usual arithmetic, see usual arithmetic conversions
void type, 46
conversion functions
multibyte/wide character, 315, 541
extended, 381, 566
restartable, 350, 382, 566
multibyte/wide string, 316, 541
restartable, 383, 567
numeric, 185, 300
wide string, 185, 370
single byte/wide character, 381
time, 345, 550
wide character, 380
conversion specifier, 276, 283, 354, 359
%A, 278, 347, 357
%B, 347
%C, 347
%D, 347
%E, 278, 356
%F, 278, 347, 356
%G, 278, 347, 356
%H, 347
%I, 347
%M, 347
%O, 347
%P, 347
%U, 348
%V, 348
%W, 348
%X, 278, 348, 356
%Y, 348
%Z, 348
%I, 285, 361
%K, 280, 286, 358, 362
%A, 278, 285, 347, 357, 361
%b, 347
%c, 279, 285, 347, 357, 361
%d, 278, 284, 347, 356, 360
%e, 278, 285, 347, 356, 361
%f, 278, 285, 356, 361
%g, 278, 285, 347, 356, 361
%h, 347
%i, 278, 356
%j, 347
%m, 347
%n, 280, 286, 347, 358, 362
%o, 278, 284, 356, 361
%p, 280, 286, 347, 358, 362
%r, 347

Index 585
%s, 279, 285, 357, 361
%t, 348
%u, 278, 284, 348, 356, 361
%w, 348
%x, 278, 284, 348, 356, 361
%y, 348
%z, 348
conversion state, 315, 350–352, 381, 382–384, 541, 566–568
conversion state functions, 381
copying functions
    string, 319, 543
    wide string, 374, 560
copysign function, 164, 226, 227, 328, 331, 417, 443, 445, 459, 464, 465, 473, 474, 515
    type-generic macro, 329
copysignl function, 227, 422, 515
copysignl128 function, 227, 422, 515
    copysignl32 function, 227, 422, 515
copysignl64 function, 227, 422, 515
    copysignl function, 227, 417, 515
copysignl type-generic macro, 329
    corresponding real type, 37
    corresponding unsigned integer type, 36
cos function, 204, 328, 414, 446, 454, 481, 515
    type-generic macro, 481
    cos function, 204, 419, 515
cosd128 function, 204, 419, 515
cosd32 function, 204, 419, 515
cosd64 function, 204, 419, 515
cosf function, 204, 414, 515
cosh function, 208, 328, 415, 447, 456, 481, 515
    type-generic macro, 328
    cosh function, 204, 414, 515
cospl function, 206, 414, 446, 455, 515
    type-generic macro, 329
    cospil function, 206, 419, 515
cospid128 function, 206, 419, 515
cospid32 function, 206, 419, 515
cospid64 function, 206, 419, 515
cospif function, 206, 414, 515
cospil function, 206, 414, 515
cpow function, 162, 328, 411, 480, 515
    type-generic macro for, 328
cpowf function, 162, 411, 515
cpowl function, 162, 331, 411, 515
cpow function, 392, 515
    cproj function, 164, 329, 331, 411, 475, 515
    cproj functions, 475
    type-generic macro, 329
cprojf function, 164, 331, 411, 515
cprojl function, 164, 331, 411, 515
cr identifier prefix, 393
    CR_DECIMAL_DIG macro, 24, 447, 516
cracos function, 393, 515
cracosh function, 393, 515
cracospi function, 393, 515
crasin function, 393, 515
crasinh function, 393, 515
crasinp function, 393, 515
cratan function, 393, 515
cratan2 function, 393, 516
cratan2pi function, 393, 516
cratanh function, 393, 516
cratanpi function, 393, 516
    ccompoundn function, 393, 516
crcos function, 393, 516
crcosh function, 393, 516
crcosp function, 393, 516
creal function, 163, 164, 165, 330, 331, 411, 473, 475, 481, 516
    type-generic macro, 475
creal type-generic macro, 329
creal type-generic macro, 481
    create function, 164, 411, 516
creall function, 164, 411, 516
    creating, 269
crexp function, 393, 516
crexp10 function, 393, 516
crexp10 function, 393, 516
crexp2 function, 393, 516
crexp2ml function, 393, 516
crexp2ml function, 393, 516
crexp2ml function, 393, 516
crexp2ml function, 393, 516
    critical undefined behavior, 570
crlog function, 393, 516
crlog function, 393, 516
crlog function, 393, 516
crlog function, 393, 516
crlog function, 393, 516
crlog function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
crlog2 function, 393, 516
    crootn function, 392, 516
croot function, 392, 516
crpow function, 393, 516
crpow function, 393, 516
crpow function, 393, 516
crpow function, 393, 516
    crsqr function, 392, 516
crsq function, 393, 516
crtan function, 393, 516
crtanh function, 393, 516
crtanpi function, 393, 516
586 Index
csin function, 159, 328, 410, 475, 516
csin functions, 475
type-generic macro for, 328
csinf function, 159, 410, 516
csinh function, 161, 328, 410, 475, 478, 516
csinh functions
type-generic macro for, 328
csinhf function, 161, 410, 516
csinhl function, 161, 410, 516
csinl function, 159, 410, 516
csinp function, 392, 516

csqrt function, 163, 328, 331, 411, 475, 480, 516
csqrt functions
type-generic macro for, 328
csqrtf function, 163, 411, 516
csqrtl function, 163, 411, 516
ctan function, 160, 328, 410, 475, 516
ctan functions, 475
type-generic macro for, 328
ctanf function, 160, 410, 516
cтанh function, 161, 328, 410, 475, 479, 516
ctanh functions
type-generic macro for, 328
ctanhf function, 161, 410, 516
ctanhl function, 161, 410, 516
ctanl function, 160, 410, 516
cotanpi function, 392, 516
cotgamma function, 392, 516
cotangent function, 345, 346, 432, 516
cotg function, xii, 346, 432, 572
cotg_s function, 432, 516, 550, 552
currency_symbol structure member, 189, 191, 193, 516

current object, 114

CX_LIMITED_RANGE pragma, viii, 146, 147, 158, 408, 410, 472, 495, 516

D format modifier, 277, 284, 356, 360
d identifier prefix, 329, 330
d-wchar sequence, 304, 372
d128 identifier prefix, 329, 393
d32 identifier prefix, 329, 330, 393
d32add macro, 31, 317, 329, 516
d32add type-generic macro, 329
d32add128 function, 232, 423, 516
d32add64 function, 232, 423, 516
d32div macro, 31, 176, 329, 516
d32div type-generic macro, 329
d32div128 function, 233, 423, 516
d32div64 function, 233, 331, 423, 516
d32fma macro, 31, 176, 329, 516
d32fma type-generic macro, 329
d32fmad128 function, 233, 423, 516
d32fmad64 function, 233, 423, 516
d32mul macro, 31, 176, 329, 516
d32mul type-generic macro, 329
d32muld128 function, 233, 423, 516
d32muld64 function, 232, 423, 516
d32sqr function, macro, 31, 176, 329, 516
d32sqr type-generic macro, 329
d32sqrtd128 function, 234, 423, 516
d32sqrtd64 function, 234, 423, 516
d32sub macro, 31, 176, 329, 516
d32sub type-generic macro, 329
d32subd128 function, 232, 331, 423, 516
d32subd64 function, 232, 423, 516
d64 identifier prefix, 327, 329, 330, 393
d64add macro, 31, 176, 329, 516
d64add type-generic macro, 329
d64add128 function, 232, 331, 423, 516
d64div macro, 31, 176, 329, 516
d64div type-generic macro, 329
d64divd128 function, 233, 423, 516
d64fma macro, 31, 176, 329, 516
d64fma type-generic macro, 329
d64fmad128 function, 233, 331, 423, 516
d64mul macro, 31, 176, 329, 516
d64mul type-generic macro, 329
d64muld128 function, 233, 423, 516
d64sqr function, macro, 31, 176, 329, 516
d64sqr type-generic macro, 329
d64sqrtd128 function, 234, 423, 516
d64sub macro, 31, 176, 329, 516
d64sub type-generic macro, 329
d64subd128 function, 232, 423, 516
dadd macro, 329, 516
dadd type-generic macro, 329
daddl function, 232, 331, 418, 442, 516
data stream, see streams
date and time header, 332, 342, 393, 550
Daylight Saving Time, 342

DBL_ identifier prefix, 392, 508
DBL_DECIMAL_DIG macro, 26, 28, 412, 438, 509
DBL_DIG macro, 26, 28, 412, 438, 509
DBL_EPSILON macro, 27, 28, 412, 439, 483, 509
DBL_HAS_SUBNORM macro, 25, 28, 412, 509
DBL_MANT_DIG macro, 25, 28, 412, 438, 483, 509
DBL_MAX macro, 27, 28, 412, 439, 483, 509
DBL_MAX_10_EXP macro, 27, 28, 412, 438, 483, 509
DBL_MAX_EXP macro, 26, 28, 412, 438, 483, 509
DBL_MIN macro, 27, 28, 412, 439, 483, 509
DBL_MIN_10_EXP macro, 26, 28, 412, 438, 493, 509
DBL_MIN_EXP macro, 26, 28, 412, 438, 483, 509
DBL_NORM_MAX macro, ii, 27, 439, 509
DBL_TRUE_MIN macro, 27, 28, 412, 509
D0 format modifier, 277, 284, 356, 360
ddiv macro, 329, 516
ddiv type-generic macro, 329
ddivl function, 233, 418, 442, 445, 516
DEC identifier prefix, 413
DEC128_ identifier prefix, 29, 392, 508
DEC128_EPSILON macro, 30, 439, 509
DEC128_MANT_DIG macro, 29, 439, 509
DEC128_MAX macro, 30, 439, 509
DEC128_MAX_EXP macro, 29, 439, 509
DEC128_MIN macro, 30, 439, 509
DEC128_TRUE_MIN macro, 29, 392, 508
DEC32_ identifier prefix, 29, 392, 508
DEC32_EPSILON macro, 30, 439, 509
DEC32_MANT_DIG macro, 29, 439, 509
DEC32_MAX macro, 30, 439, 509
DEC32_MAX_EXP macro, 29, 439, 509
DEC32_MIN macro, 30, 439, 509
DEC32_TRUE_MIN macro, 29, 392, 508
DEC64_ identifier prefix, 29, 392, 508
DEC64_EPSILON macro, 30, 439, 509
DEC64_MANT_DIG macro, 29, 439, 509
DEC64_MAX macro, 30, 439, 509
DEC64_MAX_EXP macro, 29, 439, 509
DEC64_MIN macro, 30, 439, 509
DEC64_TRUE_MIN macro, 29, 392, 508
DEC_ identifier prefix, 392, 508
DEC_EVAL_METHOD macro, 24, 29, 56, 195, 438, 502, 503, 510
DEC_INFINITY macro, 196, 234, 510
DEC_NAN macro, 196, 510
decimal constant, 52
decimal digit, 17
decimal floating types, 37
decimal re-encoding functions, 235
decimal rounding control pragma, 175
decimal signaling NaN macros, 196
decimal-point character, 152, 191
decimal128 suffix, d1 or DL, 55
decimal32 suffix, df or DF, 55
decimal64 suffix, dd or DD, 55
DECIMAL_DIG (obsolete), i, xiii, 24, 26–28, 280, 303, 358, 371, 392, 412, 438, 447, 448, 516, 572
DECIMAL_DIG macro, 26
decimal_point structure member, 189, 191, 516
declaration specifier, 87
declaration specifiers, 87
declarations, 87
  function, 108
  pointer, 105
  structure/union, 90
typedef, 111
declarator, 104
  abstract, 110
declarator type derivation, 38, 105
decodebin family, 31, 32, 237, 516
decodebind family, 237, 446, 516
decodebind128 function, 237, 423, 516
decodebind32 function, 237, 423, 516
decodebind64 function, 237, 327, 330, 423, 516
decodedec family, 31, 32, 236, 516
decodededc family, 236, 446, 516
decodededc128 function, 236, 423, 516
decodededc32 function, 236, 423, 516
decodededc64 function, 236, 327, 330, 423, 516
decrement operators, see arithmetic operators, increment and decrement
default argument promotions, 68
default initialization, 113
default label, 123, 125
DEFAULT pragma, 146, 408, 516
define preprocessing directive, 139
defined operator, 136, 137, 147, 494, 516
definition, 87
  function, 131
dependency-ordered before, 15
deprecated attribute, ii, xiv, 88, 93, 118, 119, 120, 121, 516, 572
dereference, see indirect operator
derived declarator types, 38
derived types, 37
designated initializer, 114
destringizing, 149
device input/output, 12
dfma macro, 329, 516
dfma type-generic macro, 329
dfmal function, 233, 331, 418, 442, 516
diagnostic message, 5, 10
diagnostics, 10
diagnostics header, 156
difftime function, 343, 431, 516
digit, 166
digits, 17
digraphs, 61
direct input/output functions, 295
display device, 19
div function, 300, 314, 429, 516
div_t type, 116, 300, 314, 428, 516
divide and round to narrower type, 233
division assignment operator (/=), 83
division operator (/), 40, 75, 472
dmul macro, 329, 516
dmul type-generic macro, 329
dmul1 function, 232, 418, 442, 516
documentation of implementation, 8
dot operator (.), 68
double _Complex type, 37
double _Complex type conversion, 44
double _Imaginary type, 471
double type, 36, 89
double type conversion, 44
double-precision arithmetic, 13
double-quote escape sequence ("), 58, 59, 149
double _t type, 195, 413, 450, 503, 507, 516
dsqtrt macro, 329, 516
dsqtrt _t type-generic macro, 329
dsqtrtl function, 234, 418, 442, 516
dsub macro, 329, 516
dsub _t type-generic macro, 329
dsubl function, 232, 418, 442, 516
dynamic floating-point environment, 171

E format modifier, 348
E identifier prefix, 170, 392, 508
EDOM macro, 170, 198, 411, 510, see also domain error
effective type, 64
EILSEQ macro, 170, 271, 351, 352, 367, 383–385, 411, 510, see also encoding error
element type, 37
eilf preprocessing directive, 137
ellipsis punctuator (…), 68, 108, 139
else preprocessing directive, 137
else statement, 125
empty statement, 124
encodebin family, 31, 32, 236, 516
encodebind family, 236, 237, 446, 516
encodebind128 function, 236, 423, 516
encodebind32 function, 236, 423, 516
encodebind64 function, 236, 327, 330, 423, 516
encodedec family, 31, 32, 236, 516
encodedecd family, 235, 236, 446, 516
encodedect128 function, 236, 423, 516
encodedect32 function, 235, 423, 516
encodedect64 function, 235, 327, 330, 423, 516
end-of-file, 353
end-of-file indicator, 267, 274, 292–295, 297, 298, 367, 369
end-of-file macro, see EOF macro
end-of-line indicator, 17
endif preprocessing directive, 137
enum type, 37, 89, 94
enumerated type, 37
enumeration, 37, 94
enumeration constant, 33, 56
enumeration content, 96
enumeration members, 94
enumeration specifiers, 94
enumeration tag, 34, 96
enumerator, 94
environment, 9
environment functions, 309, 538
environment list, 311, 539
environmental considerations, 17
environmental limits, 19, 240, 269, 271, 272, 280, 306, 309, 310, 358, 526
EOL macro, 510
epoch, 344
equal-sign punctuator (=), 87, 94, 112
equality operator, see equality operator
equality expressions, 78
equality operator (==), 79
ERANGE macro, 170, 185, 186, 199, 303, 304, 306, 371, 373, 374, 411, 504, 505, 510, see also range error, pole error
erf function, 220, 328, 416, 462, 516
erf _t type-generic macro, 329
erfc function, 220, 328, 416, 462, 516
erfc _t type-generic macro, 329
erfcd128 function, 220, 421, 516
erfcd32 function, 220, 421, 516
erfcd64 function, 220, 421, 516
erfclf function, 220, 416, 516
erfcl function, 220, 416, 516
erfds128 function, 220, 421, 516
erfds32 function, 220, 421, 516
erfds64 function, 220, 421, 516
erff function, 220, 416, 516
erfl function, 220, 416, 516
errange, see range error
domain, see domain error
encoding, see encoding error
pole, see pole error
range, see range error
error conditions, 198
error functions, 220, 462
error indicator, 267, 274, 292–294, 297–299, 367
error preprocessing directive, 8, 146
error-handling functions, 298, 325, 549, 550
errors header, 170, 392
escape character (\), 57
escape sequences, 17, 19, 57, 151
evaluation format, 25, 56, 195
evaluation method, 25, 65, 450
evaluation of expression, 12
evaluation order, see order of evaluation
exceptional condition, 64
excess precision, 25, 45, 130
excess range, 25, 45, 130
exclusive OR operators
     bitwise (^), 40, 80
     bitwise assignment ( ^= ), 83
executable program, 9
execution character set, 17
execution environment, 9, 10, see also environmental limits
execution sequence, 11, 123
EXIT_FAILURE macro, 300, 310, 428, 510
EXIT_SUCCESS macro, 300, 310, 338, 428, 510
exp function, 209, 211, 212, 257, 328, 331, 415, 446, 456, 516
exp type-generic macro, 328
exp10 function, 209, 210, 415, 446, 457, 516
exp10 type-generic macro, 329
exp10d128 function, 209, 420, 516
exp10d32 function, 209, 419, 516
exp10d64 function, 209, 420, 516
exp10f function, 209, 415, 516
exp10l function, 209, 415, 516
exp10ml function, 210, 415, 446, 457, 516
exp10ml type-generic macro, 329
exp10ml128 function, 210, 420, 516
exp10ml32 function, 210, 420, 516
exp10ml64 function, 210, 420, 516
exp10mlf function, 210, 415, 516
exp10mll function, 210, 415, 516
exp2 function, 210, 328, 415, 446, 457, 517
exp2 type-generic macro, 329
exp2d128 function, 210, 420, 517
exp2d32 function, 210, 420, 517
exp2d64 function, 210, 420, 517
exp2f function, 210, 415, 517
exp2l function, 210, 415, 517
exp2ml function, 210, 211, 415, 446, 457, 517
exp2ml type-generic macro, 329
exp2ml128 function, 210, 420, 517
exp2ml32 function, 210, 420, 517
exp2ml64 function, 210, 420, 517
exp2mlf function, 210, 415, 517
exp2mll function, 210, 415, 517
expd128 function, 209, 419, 517
expd32 function, 209, 419, 517
expd64 function, 209, 331, 419, 517
expf function, 209, 415, 517
expl function, 209, 415, 517
explicit conversion, 42
expml function, 211, 328, 415, 446, 457, 517
expml type-generic macro, 329
expml128 function, 211, 420, 517
expml32 function, 211, 420, 517
expml64 function, 211, 420, 517
expmlf function, 211, 415, 517
expml1 function, 211, 415, 517
exponent part, 54
exponential functions
     complex, 161, 479
     real, 209, 456
expression, 64
     assignment, 82
     cast, 75
     constant, 85
     evaluation, 12
     full, 123
     order of evaluation, see order of evaluation
     parenthesis, 65
     primary, 65
     unary, 72
     void, 46
expression statement, 124
extended alignment, 42
extended character set, 4, 17, 18
extended characters, 17
extended integer types, 36, 43, 53, 261
extended multibyte and wide character utilities header, 353, 394
extended multibyte/wide character conversion utilities, 381, 566
extended signed integer types, 36
extended unsigned integer types, 36
extensible wide character case mapping functions, 390
extensible wide character classification functions, 389
extern storage-class specifier, 34, 49, 74, 88, 98, 100, 102, 103, 106, 107, 111, 131, 132, 134, 155, 395, 402, 492, 507, 517
external definition, 131
external identifiers, underscore, 133
external linkage, 34
external name, 50
external object definitions, 133
f identifier suffix, 157, 195, 327, 329, 392, 393
fabs function, 217, 328, 331, 416, 443, 445, 460, 464, 465, 467, 473, 481, 483, 484, 517
fabs type-generic macro, 328
fabs type-generic macro, 481
fabsd128 function, 217, 421, 517
fabsd32 function, 217, 421, 517
fabsd64 function, 217, 421, 517
fabsf function, 217, 416, 483, 517
fabsl function, 217, 416, 483, 517
fadd function, 232, 418, 442, 517
fadd type-generic macro, 329
faddl function, 232, 418, 442, 517
fallthrough attribute, ii, xiv, 118, 121, 122, 572
fallthrough declaration, 121
false macro, 235, 259, 393, 426, 452, 482, 517
family
canonicalize, 32, 229
decodebin, 32, 237
decodedec, 32, 236
codebin, 32, 236
codeedec, 32, 236
modf, 32, 215, 327
strto, 32, 303
wcto, 32, 372
fclose function, 272, 273, 427, 517
fdim function, 229, 230, 328, 418, 466, 517
fdim type-generic macro, 329
fdimdb function, 229, 422, 517
fdimd32 function, 229, 422, 517
fdimf function, 229, 418, 517
fdiml function, 229, 418, 517
fdiv function, 233, 331, 418, 442, 517
fdiv type-generic macro, 329
fdivl function, 233, 418, 442, 517
FE_ identifier prefix, 172, 173, 392, 508
FE_ALL_EXCEPT macro, 84, 172, 412, 445, 463, 510
FE_DEC_DOWNWARD macro, 172, 176, 412, 446, 510
FE_DEC_DOWNWARD pragma, 172, 176, 412, 446, 510
FE_DEC_DYNAMIC pragma, 146, 176, 510
fe_dec_getround function, 172, 179, 180, 412, 446, 517
fe_dec_setround function, 172, 176, 180, 181, 412, 446, 517
FE_DEC_TONEAREST macro, 172, 173, 176, 412, 446, 510
FE_DEC_TONEAREST pragma, 146
FE_DEC_TONEARESTFROMZERO macro, 172, 176, 412, 446, 510
FE_DEC_TONEARESTFROMZERO pragma, 146
FE_DEC_TOWARDZERO macro, 172, 176, 412, 446, 510
FE_DEC_TOWARDZERO pragma, 146
FE_DEC_UPWARD macro, 172, 176, 412, 446, 510
FE_DEC_UPWARD pragma, 146
FE_DFL_ENV macro, 173, 412, 510
FE_DFL_MODE macro, 172, 180, 444, 510
FE_DIVBYZERO macro, 171, 198, 412, 485, 510
FE_DOWNWARD macro, 172, 412, 445, 510
FE_DOWNWARD pragma, 146
FE_DYNAMIC pragma, 146, 174, 175, 412, 510
FE_INEXACT macro, 171, 176, 412, 464, 510
FE_INVALID macro, 171, 179, 198, 412, 485, 510
FE_OVERFLOW macro, 172, 176, 179, 198, 412, 485, 510
FE_SNANS_ALWAYS_SIGNAL macro, 442, 444,
fexcept_t type, 171, 177, 178, 412, 495, 517
ftime function, 273, 274, 427, 497, 517
ffma function, 233, 418, 442, 517
ffma type-generic macro, 329
ffmaI function, 233, 418, 442, 517
fgetc function, 268, 271, 292, 293, 296, 427, 517
fgetpos function, 269, 296, 297, 427, 488, 498, 505, 517
fgets function, 268, 292, 293, 427, 498, 517, 536
fgetwc function, ii, 268, 270, 271, 366, 367, 368, 432, 517
fgetws function, 268, 367, 433, 498, 517
field width, 276, 354
file, 269
access functions, 272, 526
name, 270
operations, 271, 525
position indicator, 267, 269, 269, 274, 292, 293, 295–298, 367, 369
positioning functions, 296
file name, 270
file position indicator, 269
file scope, 33, 131
FILENAME_MAX macro, 267, 426, 517
finite number, 471
flags, 276, 354, see also floating-point status flag
flexible array member, 92
float _Complex type, 37
float _Complex type conversion, 44
float _Imaginary type, 471
float type, 36, 89
float type conversion, 44
float_t type, 195, 413, 450, 503, 507, 517
floating constant, 54
floating point multiply-add and round to narrower type, 233
floating suffix, f or F, 55
floating type conversion, 44, 447
floating types, 37, 151
floating-point accuracy, 24, 56, 65, 303, 447, see also contracted expression
floating-point arithmetic functions, 195, 453
floating-point classification functions, 199
floating-point control mode, 171, 450
floating-point environment, 171, 448, 450
dynamic, 171
floating-point environment header, 171, 392
floating-point exception, 171, 176, 453
floating-point number, 23, 36
floating-point rounding mode, 24
floating-point status flag, 171, 450
floor function, 197, 221, 222, 328, 417, 442, 463, 484, 517
floor type-generic macro, 329
floorl2D function, 221, 421, 517
floor32 function, 221, 421, 517
floor64 function, 221, 421, 517
floorf function, 221, 417, 517
floorl function, 221, 417, 517
FLT_ identifier prefix, 392, 508
FLT_DECIMAL_DIG macro, 26, 28, 412, 438, 510
FLT_DIG macro, 26, 28, 412, 438, 510
FLT_EPSILON macro, 27, 28, 412, 439, 483, 510
FLT_EVAL_METHOD macro, i, 24, 25, 29, 84, 85, 195, 412, 438, 470, 502, 503, 510
FLT.Has_SUBNORM macro, 25, 28, 412, 510
FLT.MANT_DIG macro, 25, 28, 412, 438, 483, 510
FLT.MAX macro, 27, 28, 412, 439, 483, 510
FLT.MAX_10_EXP macro, 27, 28, 412, 438, 510
FLT.MAX_EXP macro, 26, 28, 412, 438, 483, 510
FLT.MIN macro, 27, 28, 412, 439, 483, 510
FLT_MIN_10_EXP macro, 26, 28, 412, 438, 510
FLT_MIN_EXP macro, 26, 28, 412, 438, 483, 510
FLT.NORM_MAX macro, 27, 439, 510
FLT_RADIX macro, 24, 25, 26, 28, 29, 55, 176, 181, 215, 216, 279, 280, 303, 357, 358, 371, 412, 439, 444, 483, 510
FLT_rounds macro, 24, 25, 172, 412, 438, 441, 483, 484, 502, 510
FLT_TRUE_MIN macro, 27, 28, 412, 510
fma function, 197, 231, 328, 418, 442, 467, 517
fma type-generic macro, 329
fmad128 function, 231, 423, 517
fmad32 function, 231, 423, 517
fmad64 function, 231, 423, 517
fmaf function, 231, 418, 517
fmal function, 197, 231, 418, 517
fmax function, 230, 231, 328, 331, 418, 442, 466, 467, 473, 517
fmax type-generic macro, 329
fmaxd128 function, 230, 331, 422, 517
fmaxd32 function, 230, 422, 517
fmaxd64 function, 230, 422, 517
fmaxf function, 230, 418, 517
fmaxl function, 230, 418, 517
fmaxmag function, 230, 231, 418, 442, 467, 517
fmaxmag type-generic macro, 329
fmaxmagd128 function, 231, 423, 517
fmaxmagd32 function, 231, 422, 517
fmaxmagd64 function, 231, 423, 517
fmaxmagf function, 230, 418, 517
fmaxmagl function, 230, 418, 517
fmin function, 230, 231, 328, 418, 442, 467, 517
fmin type-generic macro, 329
fminmag function, 231, 422, 517
fminmagd32 function, 231, 422, 517
fminmagd64 function, 231, 422, 517
fminmagd128 function, 231, 422, 517
fminmagd32 function, 231, 422, 517
fminmagd64 function, 231, 422, 517
fminmagf function, 231, 418, 517
fminmagl function, 231, 418, 517
fmod function, 225, 226, 328, 417, 465, 484, 504, 517
fmod type-generic macro, 329
fmodd128 function, 226, 422, 517
fmodd32 function, 226, 422, 517
fmodd64 function, 226, 422, 517
fmodf function, 225, 417, 517
fmodl function, 225, 417, 517
fmul function, 232, 418, 442, 517
fmul type-generic macro, 329
fmull function, 232, 418, 442, 517
fopen function, 271, 272, 273, 274, 427, 497, 508, 517, 526, 527
FOPEN_MAX macro, 267, 271, 272, 426, 517, 525
fopen_s function, 427, 517, 525, 526, 527
for statement, 127
form feed, 19
form-feed character, 17, 48
form-feed escape sequence (\f), 19, 58, 168
format conversion of integer types header, 184, 392
format flag
+, 276, 354
-, 276, 354
#, 276, 355
0, 277, 355
space, 276, 355
format modifier
D, 277, 284, 356, 360
D0, 277, 284, 356, 360
E, 348
H, 277, 284, 355, 360
h, 277, 284, 355, 360
hh, 277, 284, 355, 360
j, 277, 284, 355, 360
L, 277, 284, 355, 360
l, 277, 284, 355, 360
ll, 277, 284, 355, 360
0, 348
t, 277, 284, 355, 360
z, 277, 284, 355, 360
formatted input/output functions, 190, 275, 528
wide character, 354, 553
forward reference, 5
FP_. identifier prefix, 196, 392, 508
FP_. CONTRACT pragma, viii, 65, 146, 147, 199, 408, 414, 473, 495, 502, 510, see also contracted expression
FP_FAST_D32ADD128 macro, 197, 510
FP_FAST_D32ADD64 macro, 197, 510
FP_FAST_D32DIV128 macro, 197, 510
FP_FAST_D32DIV64 macro, 197, 510
FP_FAST_D32FMAD128 macro, 197, 510
FP_FAST_D32FMAD64 macro, 197, 510
FP_FAST_D32MUL128 macro, 197, 510
FP_FAST_D32MUL64 macro, 197, 510
FP_FAST_D32SQR128 macro, 197, 510
FP_FAST_D32SQR64 macro, 197, 510
FP_FAST_D32SUB128 macro, 197, 510
FP_FAST_D32SUB64 macro, 197, 510
FP_FAST_D64ADD128 macro, 197, 510
FP_FAST_D64DIV128 macro, 197, 510
FP_FAST_D64FMAD128 macro, 197, 510
FP_FAST_D64MUL128 macro, 197, 510
FP_FAST_D64MUL64 macro, 197, 510
FP_FAST_D64SQR128 macro, 197, 510
FP_FAST_D64SQR64 macro, 197, 510
FP_FAST_DADDL macro, 197, 198, 510
FP_FAST_DDIVL macro, 197, 510
FP_FAST_DFMAL macro, 197, 510
FP_FAST_DMULL macro, 197, 510
FP_FAST_DSQRTL macro, 197, 510
FP_FAST_DSUBL macro, 197, 510
FP_FAST_FADD macro, 197, 198, 510
FP_FAST_FADDL macro, 197, 198, 510
FP_FAST_FDIV macro, 197, 510
FP_FAST_FDIVL macro, 197, 510
FP_FAST_FFMA macro, 197, 510
FP_FAST_FFMAL macro, 197, 510
FP_FAST_FMA macro, 197, 414, 510
FP_FAST_FMAD128 macro, 197, 510
FP_FAST_FMAD32 macro, 197, 510
FP_FAST_FMAD64 macro, 197, 510
FP_FAST_FMAF macro, 197, 414, 510
FP_FAST_FMAU macro, 197, 414, 510
FP_FAST_FMUL macro, 197, 510
FP_FAST_FNULL macro, 197, 510
FP_FAST_FSQRT macro, 197, 510
FP_FAST_FSQRTL macro, 197, 510
FP_FAST_FSUB macro, 197, 510
FP_FAST_FSUBL macro, 197, 510
FP_.ilogb macro, 198, 212, 414, 510
FP_.ilogbNan macro, 198, 212, 414, 510
FP_.INFINITY macro, 196, 413, 510
FP_.INT..DOWNWARD macro, 197, 510
FP_.INT_.TONEAREST macro, 197, 510
FP_.INT_.TONEARESTFROMZERO macro, 197, 510
FP_.INT_.TOWARDZERO macro, 197, 510
FP_.INT_.UPWARD macro, 197, 225, 510
FP_.LLOGB macro, 198, 212, 510

Index 593
function, 197, fromfp
frexp function, 267, 269, 296, 297, 426, 427, 517
fprintfs function, 427, 517, 528
fputc function, 19, 268, 271, 293, 294, 296, 427, 517
fputs function, 144, 268, 293, 427, 517
fputwc function, 268, 271, 367, 369, 433, 517
fputws function, 268, 367, 368, 433, 517
frac_digits structure member, 189, 191, 193, 198, 212, 510
frexp function, 211, 328, 415, 457, 488, 489, 498, 517
frexp type-generic macro, 329
frexpd128 function, 211, 420, 517
frexpd32 function, 211, 420, 517
frexpd64 function, 211, 420, 517
frexpf function, 211, 415, 517
frexpl function, 211, 415, 517
fromfp function, 197, 224, 225, 417, 443, 447, 464, 517
fromfpl functions, 224
fromfp type-generic macro, 329
fromfpd128 function, 224, 422, 517
fromfpd32 function, 224, 422, 517
fromfpd64 function, 224, 422, 517
fromfpf function, 224, 417, 517
fromfpfl function, 224, 417, 517
fromfpfx function, 197, 225, 417, 443, 447, 464, 517
fromfpfx functions, 225
fromfp type-generic macro, 329
fromfpf128 function, 225, 422, 517
fromfpf32 function, 225, 422, 517
fromfpf64 function, 225, 422, 517
fromfpfl function, 225, 417, 517
fscanf function, 184, 268, 282, 283, 286–290, 393, 427, 504, 517, 529
fscanf function, 428, 517, 528, 529, 530, 532, 533
fseek function, 268, 270, 274, 295, 297, 298, 369, 427, 498, 517
fsetpos function, 269, 274, 295, 296, 297, 369, 427, 498, 505, 517
fsqrt function, 234, 418, 442, 517
fsqrt type-generic macro, 329
fsqrtl function, 234, 418, 442, 517
fsub function, 232, 418, 442, 517
fsub type-generic macro, 329
fsubl function, 232, 331, 418, 442, 517
fseek function, 297, 298, 427, 489, 498, 505, 517
full declarator, 105
full expression, 123
fully buffered, 270
fully buffered stream, 270
function
argument, 67, 132
body, 131
call, 67
library, 154
declarator, 107, 151
definition, 108, 131
designator, 46
image, 19
inline, 102
library, 9, 154
name length, 20, 50, 151
no-return, 102
parameter, 11, 67, 88, 132
prototype, 11, 33, 41, 68, 107, 132, 151, 152, 195
prototype scope, 33, 106, 107
recursive call, 68
return, 129, 448
scope, 33
type, 38
type conversion, 46
function prototype scope, 33
function scope, 33
function specifiers, 102
function type, 36
function-call operator (()), 67
function-like macro, 139
fundamental alignment, 41
future directions
language, 151
library, 392
fwrite function, 269, 368, 433, 517
fwrite function, 184, 268, 354, 358, 359, 362–364, 366, 394, 432, 500, 504, 518, 553
fwrite function, 434, 518, 553
fwrite function, 268, 296, 427, 498, 518
fwsprintf function, 184, 268, 359, 362–364, 366, 394, 432, 504, 518, 554
fwsprintf function, 434, 518, 553
fwsprintf function, 434, 518, 553, 554, 556, 557, 560
gamma functions, 220, 462
general utilities, 537
wide string, 370, 560
general utilities header, 300, 393
general wide string utilities, 370, 560
generic association, 66
generic parameters, 327
generic selection, 65
getc function, 268, 293, 294, 427, 518
getchar function, 14, 268, 294, 427, 518
getenv function, 311, 428, 499, 500, 518
getenv_s function, 429, 518, 538, 539
getpayload function, 424, 445, 468, 469, 518
getpayload128 function, 424, 469, 518
getpayload32 function, 424, 469, 518
getpayload64 function, 424, 469, 518
getpayloadd64 function, 424, 468, 518
gets function, 428, 468, 518
gets_s function, 428, 518, 536
getwc function, 268, 368, 433, 518
getwchar function, 268, 368, 369, 433, 518
gmtime function, 345, 346, 432, 518
gmtime_r function, xiii, 346, 432, 572
gmtime_s function, 432, 518, 552
go statement, 33, 124
graphic characters, 17
greater-than operator (>, 78
greater-than-or-equal-to operator (>=), 78
grouping structure member, 189, 191, 192, 518

H format modifier, 277, 284, 355, 360
h format modifier, 277, 284, 355, 360
happens before, 16
header, 9, 152, see also standard headers
header names, 48, 61, 138
hexadecimal constant, 52
hexadecimal digit, 52, 55, 58
\hexadecimal{digits} (hexadecimal-character escape sequence), 58
hexadecimal prefix, 52
hexadecimal-character escape sequence (\x
\hexadecimal{digits}), 58
hh format modifier, 277, 284, 355, 360
hidden, 33
high-order bit, 4
horizontal tab, 19
horizontal-tab character, 17, 48
horizontal-tab escape sequence (\t), 19, 58, 166, 168, 387
hosted, 10
hosted execution environment, 8, 10
HUGE_VAL macro, 195, 199, 228, 229, 303, 371, 413, 453, 518
HUGE_VAL_D128 macro, 195, 518
HUGE_VAL32 macro, 195, 196, 518
HUGE_VAL_D64 macro, 195, 518
HUGE_ValF macro, 195, 199, 303, 371, 413, 453, 518
HUGE_ValL macro, 195, 199, 303, 371, 413, 453, 518
hyperbolic functions
complex, 160, 476
real, 207, 456
hypot function, 217, 328, 416, 442, 446, 460, 475, 518
hypot type-generic macro, 329
hypotd128 function, 217, 421, 518
hypotd32 function, 217, 421, 518
hypotd64 function, 217, 421, 518
hypotf function, 217, 416, 518
hypotl function, 217, 416, 518
I macro, 518
I macro, 164, 475
identifier, 50, 65
linkage, see linkage
maximum length, 50
name spaces, 34
reserved, 49, 153, 508, 514, 523
rules, 508
scope, 33
type, 36
identifier list, 136
identifier nondigit, 49
IEC 60559, 12, 171, 181, 237
IEC 60559, 2, 23, 24, 149, 157, 199, 226, 440, 471, 482
IEEE floating-point arithmetic standard, see
IEC 60559, ANSI/IEEE 754, ANSI/IEEE 854
IEEE 754, 440
IEEE 854, 440
if preprocessing directive, 21, 24, 137, 154
ifdef, 8, 135, 137, 175, 179, 180, 202–231, 232, 233, 234–237, 301, 303, 331, 371, 407, 438, 439, 468, 469, 518
ifdef preprocessing directive, 137
ifndef preprocessing directive, 137
ignore_handler_s function, 429, 518, 538
ilogb function, 198, 211, 212, 328, 415, 442, 457, 458, 488, 518
ilogb type-generic macro, 329
ilogbd128 function, 212, 420, 518
ilogbd32 function, 212, 420, 518
ilogbd64 function, 212, 420, 518
ilogbf function, 211, 415, 518
ilogbl function, 211, 415, 518
imaginary macro, 157, 410, 474, 475, 518
imaginary numbers, 471
imaginary type domain, 471
imaginary types, 471
imaxabs function, 185, 413, 518
imaxdiv function, 184, 185, 413, 518
imaxdiv_t type, 184, 185, 413, 518
implementation, 5
implementation limit, 5, 8, 20, 50, 105, 126, 438,
see also environmental limits
implementation-defined behavior, 3, 8, 500
implementation-defined value, 6
implicit conversion, 42
implicit initialization, 113
include preprocessing directive, 9, 138
inclusive OR operators
  bitwise (|), 40, 80
  bitwise assignment (|=), 83
incomplete, 36
incomplete type, 36
increment operators, see arithmetic operators, increment and decrement
indeterminately sequenced, 12, 68, 70, 83, see also sequenced before, unsequenced
indirection operator (*), 67, 73
inequality operator (!=), 79
infinity, 471
INFINITY macro, 164, 196, 278, 302–304, 356, 370–372, 413, 441, 473, 474, 518
initial position, 19
initial shift state, 18
initialization, 10, 35, 46, 71, 112, 449
  in blocks, 123
initialized, 10
initializer, 112
  permitted form, 85
  string literal, 46
inline, 102
  inline definition, 102
  inline function, 102
inner scope, 33
input failure, 365, 366, 529, 530, 532–534, 536, 554, 556–560
input/output functions
  character, 292, 536
  direct, 295
  formatted, 275, 528
  wide character, 354, 553
  wide character, 366
  formatted, 354, 553
input/output header, 267, 393, 524
input/output, device, 12
INT identifier prefix, 263, 266, 393, 426, 508
int identifier prefix, 261, 393, 426, 508
int type, 36, 43, 53, 89
int type conversion, 43, 44
intN_t types, 261
INTN_t types, 261
INTN_C macros, 266
INTN_MAX macros, 263
INTN_MIN macros, 263
INTN_WIDTH macros, 263
INT16_C macro, 511
INT16_MAX macro, 511
INT16_MIN macro, 511
int16_t type, 511
INT32_C macro, 511
INT32_MAX macro, 511
INT32_MIN macro, 511
int32_t type, 511
INT64_C macro, 511
INT64_MAX macro, 511
INT64_MIN macro, 511
int64_t type, 511
INT8_C macro, 511
INT8_MAX macro, 511
INT8_MIN macro, 511
int8_t type, 261, 511
INT_FAST identifier prefix, 264, 426
int_fast identifier prefix, 262, 426
INT_LEAST identifier prefix, 263, 426
int_least identifier prefix, 261, 266, 426
int_curr_symbol structure member, 189, 192, 193, 518
INT_FASTN_MAX macros, 264
INT_FASTN_MIN macros, 264
INT_FASTN_WIDTH macros, 264
int_fast16_t type, 255, 262, 511
int_fast32_t type, 184, 255, 262, 511
int_fast64_t type, 255, 262, 511
int_fast8_t type, 255, 262, 511
int_fastN_t types, 262
int_frac_digits structure member, 189, 192, 193, 518
INT_LEASTN_MAX macros, 263
INT_LEASTN_MIN macros, 263
int_leastN_t types, 261
INT_LEASTN_WIDTH macros, 263
int_least16_t type, 254, 262, 511
int_least32_t type, 255, 261, 262, 511
int_least64_t type, 255, 262, 511
int_least8_t type, 254, 262, 511
INT_MAX macro, 22, 36, 137, 198, 212, 413, 438, 482, 497, 498, 511
INT_MIN macro, 22, 36, 198, 413, 438, 482, 511
int_n_cs_precedes structure member, 189, 192, 193, 518
int_n_sep_by_space structure member, 189, 192, 193, 518
int_n_sign_posn structure member, 189, 192, 193, 518
int_p_cs_precedes structure member, 189, 192, 193, 518
int_p_sep_by_space structure member, 189, 192, 193, 518
int_p_sign_posn structure member, 189, 192, 193, 518
INT_MIN macro, 22, 36, 198, 413, 438, 482, 511
INT_WIDTH macro, 22, 225, 511
integer arithmetic functions, 185, 314
integer character constant, 57
integer constant, 52
integer constant expression, 46, 85, 91, 95, 106, 113, 117, 125, 136, 154
integer conversion rank, 42
integer promotions, 13, 21, 43, 68, 73, 77, 125, 263, 265, 277, 355
integer suffix, 53
integer type conversion, 42, 43, 44, 447
integer types, 37, 261
  extended, 36, 43, 53, 261
integer types header, 261, 393
inter-thread happens before, 15
interactive device, 12, 270, 274
internal linkage, 34
internal name, 50
interrupt, 19
INTMAX_C macro, 266, 426, 511
INTMAX_MAX macro, 185, 186, 264, 426, 511
INTMAX_MIN macro, 185, 186, 264, 426, 511
intmax_t type, 137, 185, 186, 216–219, 224, 225, 256, 262, 266, 277, 284, 355, 360, 413, 416, 417, 421, 422, 426, 511, 573
INTMAX_WIDTH macro, 265, 426, 511
INTPTR_MAX macro, 264, 426, 511
INTPTR_MIN macro, 264, 426, 511
intptr_t type, 255, 262, 426, 511
is identifier prefix, 392, 394, 508
isalnum function, 166, 167, 168, 411, 511
isalpha function, 166, 386, 411, 506, 511
isblank function, 166, 411, 506, 511
iscanonical macro, 24, 200, 414, 444, 511
iscntrl function, 166, 167, 168, 411, 511
isdigit function, 166, 167, 168, 190, 411, 511
iseqsig macro, 239, 418, 443, 470, 511
isfinite macro, 200, 414, 443, 444, 473, 474, 511
isgraph function, 167, 386, 411, 511
isgreater macro, 237, 238, 418, 443, 467, 511
isgreateerequal macro, 238, 418, 443, 452, 466, 511
isinf macro, 200, 201, 414, 444, 459, 473, 474, 511
isless macro, 238, 418, 443, 452, 511
islessequal macro, 238, 418, 443, 511
islessgreater macro, 238, 239, 418, 444, 511
islower function, 4, 166, 167, 168, 190, 411, 506, 511
isnan macro, 201, 414, 444, 466, 473, 474, 511
isnormal macro, 201, 414, 443, 444, 511
ISO/IEC 10976–1, 37, 64
ISO/IEC 646, 18
ISO/IEC 9945–2, 189
ISO/IEC 10646, 2, 50, 51, 148
ISO/IEC 10976–1, 482
ISO/IEC 2382, 2, 3
ISO 4217, 2, 192
ISO 80000–2, 2, 3
ISO 8601, 2, 347
isprint function, 19, 167, 411, 511
ispunct function, 166, 167, 168, 411, 506, 511
issignaling macro, 201, 202, 414, 443, 444, 511
isspace function, 152, 166, 167, 168, 411, 506, 511
issubnormal macro, 202, 414, 444, 511
isunordered macro, 239, 418, 443, 444, 511
isupper function, 166, 168, 169, 411, 506, 511
iswalnum function, 387, 388, 389, 434, 511
iswalpha function, 386, 387, 389, 434, 506, 511
iswblank function, 387, 389, 434, 506, 511
iswcntrl function, 387, 388, 389, 434, 511
iswctype function, 389, 390, 435, 500, 506, 511
iswdigit function, 387, 388, 389, 434, 511
iswgraph function, 386, 388, 389, 434, 511
iswlower function, 387, 388, 389, 390, 434, 506, 511
iswprint function, 386, 388, 389, 434, 511
iswpunct function, 386, 387, 388, 389, 434, 506, 511
iswspace function, 152, 386, 387, 388, 389, 435, 506, 511
iswupper function, 387, 388, 389, 390, 435, 506, 511
iswxdigit function, 389, 435, 511
isxdigit function, 168, 190, 411, 511
iszero macro, 202, 414, 444, 511
italic type convention, 3, 33
iteration statements, 127
j format modifier, 277, 284, 355, 360
jmp_buf type, 240, 241, 424, 518, 571
jump statements, 128

keyword
  _Alignas, 41, 49, 103, 104, 245, 395, 403, 508
  _Alignof operator, 41, 42, 46, 49, 72, 74, 85, 104, 106, 131, 245, 395, 400, 492, 505, 508
  _Complex types, 25, 37, 45, 49, 89, 90, 157, 395, 402, 471–473, 509
  _Generic, 49, 65, 66, 330, 395, 400, 511
  _Static_assert, xiii, 49, 117, 156, 395, 405, 512, 572
  _Thread_local storage-class specifier, 35, 49, 88, 107, 332, 395, 402, 513

Index 597
auto storage-class specifier, 49, 88, 107, 127, 131, 395, 402, 515
continue, 49, 128, 129, 395, 406, 515
defined operator, 136, 137, 147, 494, 516
extern storage-class specifier, 34, 49, 74, 88, 98, 100, 102, 103, 106, 107, 111, 131, 132, 134, 155, 395, 402, 492, 507, 517
ifdef, 8, 135, 137, 175, 179, 180, 202–231, 232, 233, 234–237, 301, 303, 331, 371, 407, 438, 439, 468, 469, 518
undef, 135, 136, 142, 147, 154, 155, 407, 494, 521

keywords, 49, 471, 507
kill_dependency macro, 15, 252, 253, 425, 518

known constant size, 38

l encoding prefix, 57–60, 398
\ encoding prefix, 57–60, 398
l format modifier, 277, 284, 355, 360
\ format modifier, 277, 284, 355, 360
\ identifier suffix, 157, 195, 327, 330, 392, 393
Ltmpnam macro, 267, 272, 426, 519
Ltmpnam_s macro, 427, 519, 524, 525
label name, 33, 34
labeled statement, 123
labs function, 314, 428, 482, 518
language, 33

encoding prefix
L, 57–60, 398
U, 57–60, 398
u, 57–60, 398
u8, 57–59, 398
future directions, 151
syntax summary, 395

Latin alphabet, 17, 50
LC_ identifier prefix, 189, 392, 508
LC_ALL macro, 189, 190, 193, 413, 511
LC_COLLATE macro, 189, 190, 322, 376, 413, 511
LC_CTYPE macro, 189, 190, 300, 315, 316, 381, 386, 389–391, 413, 499, 500, 511, 541
LC_MONETARY macro, 189, 190, 193, 413, 511
LC_NUMERIC macro, 189, 190, 193, 413, 511
LC_TIME macro, 189, 190, 345, 347, 413, 511
lconv structure type, 189, 190, 413, 518
LDBL_ identifier prefix, 392, 508
LDBL_DECIMAL_DIG macro, 26, 392, 412, 439, 511
LDBL_DIG macro, 26, 412, 439, 511
LDBL_EPSILON macro, 27, 412, 439, 483, 511
LDBL_HAS_SUBNORM macro, 25, 412, 511
LDBL_MANT_DIG macro, 25, 412, 439, 483, 511
LDBL_MAX macro, 27, 412, 439, 483, 511
LDBL_MAX_10_EXP macro, 27, 412, 439, 511
LDBL_MAX_EXP macro, 26, 412, 439, 483, 511
LDBL_MIN macro, 27, 412, 439, 483, 511
LDBL_MIN_10_EXP macro, 26, 412, 439, 511
LDBL_MIN_EXP macro, 26, 412, 439, 483, 511
LDBL_NORM_MAX macro, 27, 439, 511
LDBL_TRUE_MIN macro, 27, 412, 511
ldexp function, 212, 328, 415, 458, 518
ldexp type-generic macro, 329
ldexpd128 function, 212, 420, 518
ldexpd32 function, 212, 420, 518
ldexpd64 function, 212, 420, 518
ldexpf function, 212, 415, 518
ldexpl function, 212, 415, 518
tdiv function, 300, 314, 429, 518
tdiv_t type, 300, 314, 428, 429, 518
leading underscore in identifiers, 153
left-shift assignment operator (<<), 83
left-shift operator (<<), 40, 77

length

external name, 20, 50, 151
function name, 20, 50, 151
identifier, 50
internal name, 20, 50
length function, 315, 326, 380, 382, 550, 565
length modifier, 276, 283, 354, 359
length of a string, 152
length of a wide string, 152
less-than operator (<), 78
less-than-or-equal-to operator (<=), 78
letter, 18, 166

lexical elements, 9, 48
lgamma function, 220, 221, 328, 416, 462, 518
lgamma type-generic macro, 329
lgammal28 function, 220, 421, 518
lgammad32 function, 220, 421, 518
lgammad64 function, 220, 421, 518
lgammaf function, 220, 416, 518
lgammal function, 220, 416, 518
library, 9, 152, 522
constant

memory_order_acquire, 250, 251–253, 255, 256, 258, 425, 512
memory_order_acquire, 250, 251, 253, 255, 258, 425, 512
memory_order_consume, 250, 251, 253, 255, 425, 512
memory_order_relaxed, 250, 251–253, 425, 512
memory_order_release, 250, 251, 253, 255, 256, 425, 512
memory_order_seq_cst, 17, 39, 70, 83, 249, 250, 251, 253, 425, 512
mtx_plain function, 332, 336, 431, 512
mtx_recursive function, 333, 336, 431, 512
atomic_compare_exchange_strong_explicit
  256, 425, 508
atomic_compare_exchange_weak,
  256, 257, 425, 508
atomic_compare_exchange_weak_explicit
  256, 425, 508
atomic_exchange, 256, 425, 508
atomic_exchange_explicit, 256,
  425, 509
atomic_fetch_, 257, 425, 509
atomic_fetch_add, 509
atomic_fetch_add_explicit, 509
atomic_fetch_and, 509
atomic_fetch_and_explicit, 509
atomic_fetch_or, 509
atomic_fetch_or_explicit, 509
atomic_fetch_sub, 509
atomic_fetch_sub_explicit, 509
atomic_fetch_xor, 509
atomic_fetch_xor_explicit, 509
atomic_flag_clear, 258, 425, 509
atomic_flag_clear_explicit, 258,
  426, 509
atomic_flag_test_and_set, 258,
  425, 509
atomic_flag_test_and_set_explicit,
  258, 425, 509
atomic_init, 250, 425, 509
atomic_is_lock_free, 243, 254, 425,
  496, 509
atomic_load, 255, 257, 425, 509
atomic_load_explicit, 252, 255, 425,
  509
atomic_signal_fence, 253, 254, 425,
  509
atomic_store, 255, 425, 509
atomic_store_explicit, 252, 255,
  425, 509
atomic_thread_fence, 253, 425, 509
bsearch, 312, 313, 428, 489, 499, 515
bsearch_s, 429, 515, 539, 540
btowc, 357, 359, 381, 433, 515
c16rtomb, i, 351, 432, 515
c32rtomb, 352, 432, 515
cabs, 162, 328, 411, 475, 515
cabsf, 162, 321, 411, 515
cabsl, 162, 411, 515
cacos, 158, 328, 410, 475, 476, 515
cacosf, 158, 410, 515
cacosl, 160, 328, 410, 476, 515
cacoslf, 160, 410, 515
cacosl, 158, 410, 515
cacosp, 392, 515
call_once, 332, 333, 431, 515
calloc, 307, 308, 428, 489, 498, 505, 515
canonicalize, 24, 229, 328, 418, 443,
  466, 467, 515
canonicalized128, 229, 422, 515
canonicalized32, 229, 422, 515
canonicalized64, 229, 422, 515
canonicalizef, 229, 418, 515
canonicalizedl, 229, 418, 515
carg, 163, 329, 331, 411, 475, 481, 515
cargv, 163, 411, 515
cargl, 163, 411, 515
casin, 159, 328, 410, 475, 515
casinl, 159, 410, 515
casinh, 160, 328, 410, 475, 477, 515
casinhf, 160, 410, 515
casinhl, 160, 410, 515
casinpi, 392, 515
catan, 159, 328, 410, 475, 515
catanlg, 159, 410, 515
catanh, 160, 161, 328, 410, 475, 477, 515
catanhf, 160, 410, 515
catanhl, 160, 410, 515
catanl, 159, 410, 515
catandp, 392, 515
ccbrt, i, 66, 216, 328, 330, 416, 459, 515
ccbrtd128, 216, 421, 515
ccbrtd32, 216, 420, 515
ccbrtd64, 216, 420, 515
ccbrtf, 66, 216, 330, 416, 515
ccbtl, 66, 216, 330, 416, 515
ccompoundn, 392, 515
ccos, 159, 328, 410, 475, 515
ccosf, 159, 410, 515
ccosh, 161, 328, 410, 475, 478, 515
ccoshf, 161, 410, 515
ccoshl, 161, 410, 515
ccosl, 159, 410, 515
ccospi, 392, 515
ceil, 197, 221, 328, 331, 416, 442, 462,
  463, 464, 484, 515
ceilid128, 32, 221, 421, 515
ceilid32, 32, 221, 421, 515
ceilid64, 32, 221, 421, 515
ceilf, 221, 416, 515
ceill, 221, 225, 417, 515
cerf, 392, 515
cerfc, 392, 515
ceph, 161, 162, 328, 410, 479, 480, 515
cepl, 392, 515
cepl0, 392, 515
cepl0m, 392, 515
cepl2, 392, 515
cepl2m, 392, 515
ceplf, 161, 410, 515
cepll, 161, 410, 515
ceplm, 392, 515
cepxm, 392, 515
cimag, 163, 164, 329, 331, 411, 473, 475, 481, 515
cimagf, 163, 411, 515
cimagl, 163, 331, 411, 515
clearerr, 298, 427, 515
clgamma, 392, 515
clock, 342, 343, 431, 505, 515
clog, 162, 328, 410, 480, 515
clog10, 392, 515

clog10p1, 392, 515
clog1p, 392, 515
clog2, 392, 515

clog2p1, 392, 515
clogf, 162, 331, 410, 515
clogl, 162, 411, 515
clogpl, 392, 515
cnd_broadcast, 333, 334, 335, 431, 509
cnd_destroy, 334, 431, 509
cnd_init, 334, 431, 509
cnd_signal, 334, 335, 431, 509
cnd_timedwait, 334, 335, 431, 509
cnd_wait, 334, 335, 431, 509

compoundnd, 216, 217, 416, 446, 459, 515

compoundnd128, 217, 421, 515

compoundnd32, 216, 421, 515

compoundnd64, 217, 421, 515

compoundndn, 216, 416, 515

conj, 164, 329, 411, 475–480, 515

conjf, 164, 411, 515

conjl, 164, 411, 515


copysignd128, 227, 422, 515


copysignd32, 227, 422, 515

copysignnd4, 227, 422, 515

copysignfn, 227, 417, 515

copysignl, 227, 331, 417, 515

cos, 204, 328, 414, 446, 454, 481, 515

cosd128, 204, 419, 515

cosd32, 204, 419, 515


cosd64, 204, 419, 515

cosf, 204, 414, 515

cosh, 208, 328, 415, 447, 456, 481, 515


coshd128, 208, 419, 515

coshd32, 208, 419, 515

coshd64, 208, 419, 515

coshf, 208, 415, 515


coshl, 208, 415, 515

cosl, 204, 414, 515

cosp, 206, 414, 446, 455, 515

cospid128, 206, 419, 515

cospid32, 206, 419, 515

cospid64, 206, 419, 515

cospif, 206, 414, 515

cospil, 206, 414, 515

cp, 162, 328, 411, 480, 515

cp, 162, 411, 515

cp, 162, 331, 411, 515

cp, 392, 515

cp, 392, 515

cp, 164, 329, 331, 411, 475, 515

cp, 164, 331, 411, 515


cp, 164, 331, 411, 515


cracos, 393, 515

cracosf, 393, 515


cracospi, 393, 515

crasin, 393, 515

crasinh, 393, 515

crasinif, 393, 515

cratan, 393, 515

cratan2, 393, 516


cratan2pi, 393, 516


cratan2pi2, 393, 516


cratanh, 393, 516

cratanpi, 393, 516


crcompoundn, 393, 516

crcos, 393, 516

crcosh, 393, 516

crcospi, 393, 516

creal, 163, 164, 165, 330, 331, 411, 473, 475, 481, 516

crealf, 164, 411, 516

creal16, 164, 411, 516

crealf, 393, 516

crexp, 393, 516

crexp10, 393, 516

crexp10ml, 393, 516

crexp2, 393, 516

crexp2ml, 393, 516

crexp2n, 393, 516

crexpml, 393, 516

crhypot, 393, 516

crlog, 393, 516

crlog10, 393, 516


crlog10p1, 393, 516


crloga, 393, 516

crlog2, 393, 516

crlog2p1, 393, 516


crlogpl, 393, 516

crotn, 392, 516

crpow, 393, 516

crpow, 393, 516

crpowr, 393, 516

crrootn, 393, 516

crrootn, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

crsqrt, 393, 516

csin, 159, 328, 410, 475, 516

csinf, 159, 410, 516

Index 601
encodedc32, 328, 410, 475, 478, 516
encodedc64, 328, 330, 423, 516
encodedc128, 220, 421, 516
erf, 220, 328, 416, 462, 516
erfc, 220, 328, 416, 462, 516
erfcd128, 220, 421, 516
erfcd32, 220, 421, 516
erfcd64, 220, 421, 516
erfcl, 220, 416, 516
erfd128, 220, 421, 516
erfd32, 220, 421, 516
erfd64, 220, 421, 516
erff, 220, 416, 516
erfl, 220, 416, 516
exp, 209, 211, 212, 257, 328, 331, 415, 446, 456, 516
exp10, 209, 210, 415, 446, 457, 516
exp10d128, 209, 420, 516
exp10d32, 209, 419, 516
exp10d64, 209, 420, 516
exp10f, 209, 415, 516
exp10l, 209, 415, 516
exp10m1, 210, 415, 446, 457, 516
exp10m1d128, 210, 420, 516
exp10m1d32, 210, 420, 516
exp10m1d64, 210, 420, 516
exp10m1f, 210, 415, 516
exp10m1l, 210, 415, 516
exp2, 210, 328, 415, 446, 457, 517
exp2d128, 210, 420, 517
exp2d32, 210, 420, 517
exp2d64, 210, 420, 517
exp2f, 210, 415, 517
exp2l, 210, 415, 517
exp2m1, 210, 211, 415, 446, 457, 517
exp2m1d128, 210, 420, 517
exp2m1d32, 210, 420, 517
exp2m1d64, 210, 420, 517
exp2m1f, 210, 415, 517
exp2m1l, 210, 415, 517
expd128, 209, 419, 517
expd32, 209, 419, 517
expd64, 209, 331, 419, 517
expf, 209, 415, 517
expl, 209, 415, 517
expml, 211, 328, 415, 446, 457, 517
expml1d128, 211, 420, 517
expml1d32, 211, 420, 517
expml1d64, 211, 420, 517
expml1f, 211, 415, 517
expml1l, 211, 415, 517
fabs, 217, 328, 331, 416, 443, 445, 460, 464, 465, 467, 473, 481, 483, 484, 517
fabsd128, 217, 421, 517
fabsd32, 217, 421, 517
fabsd64, 217, 421, 517
fabsf, 217, 416, 483, 517
Index
Index

ISO/IEC 9899:202x (E)  working draft — November 18, 2019  N2455

fopen, 269, 274, 275, 427, 517
fopen_s, 427, 517, 527, 528
frexp, 211, 328, 415, 457, 488, 489, 517
frexp128, 211, 420, 517
frexp32, 211, 420, 517
frexp64, 211, 420, 517
frexpf, 211, 415, 517
frexpl, 211, 415, 517
fromfp, 197, 224, 225, 417, 443, 447, 464, 517
fromfpd128, 224, 422, 517
fromfpd32, 224, 422, 517
fromfpd64, 224, 422, 517
fromfpf, 224, 417, 517
fromfpl, 224, 417, 517
fromfp, 197, 225, 417, 443, 447, 464, 517
fromfpd128, 225, 422, 517
fromfpd32, 225, 422, 517
fromfpd64, 225, 422, 517
fromfpfx, 225, 417, 517
fromfpzl, 225, 417, 517
fscanf, 184, 268, 282, 283, 286–290, 393, 427, 504, 517, 529
fscanf_s, 428, 517, 528, 529, 530, 532, 533
fseek, 268, 270, 274, 295, 297, 298, 369, 427, 498, 517
fsetpos, 269, 274, 295, 297, 298, 369, 427, 498, 505, 517
fsqrt, 234, 418, 442, 517
fsqrtl, 234, 418, 442, 517
fsub, 232, 418, 442, 517
fsubl, 232, 331, 418, 442, 517
ftell, 297, 298, 427, 489, 498, 505, 517
fwrite, 269, 368, 433, 517
fwrite, 184, 268, 354, 358, 359, 362–364, 366, 369, 432, 504, 518, 554
fwrite, 434, 518, 553
fwrite, 268, 296, 427, 498, 518
fwscanf, 184, 268, 359, 362–364, 366, 369, 432, 504, 518, 554
fwscanf_s, 434, 518, 553, 554, 556, 557, 560
getc, 268, 293, 294, 427, 518
getchar, 14, 268, 294, 427, 518
getenv, 311, 428, 499, 500, 518
getenv_s, 429, 518, 538, 539
getpayload, 424, 445, 468, 469, 518
getpayload128, 424, 469, 518
getpayload32, 424, 469, 518
getpayload64, 424, 469, 518
getpayloadf, 424, 468, 518
getpayloadl, 424, 468, 518
gets_s, 428, 518, 536
getwc, 268, 368, 433, 518
getwchar, 268, 368, 369, 433, 518
gmtime, 345, 346, 432, 518
gmtime_r, xii, 346, 432, 572
gmtime_s, 432, 518, 552
hypot, 217, 328, 416, 442, 446, 460, 475, 518
hypotd128, 217, 421, 518
hypotd32, 217, 421, 518
hypotd64, 217, 421, 518
hypotf, 217, 416, 518
hypotl, 217, 416, 518
ignore_handler_s, 429, 518, 538
ilogb, 198, 211, 212, 328, 415, 442, 457, 458, 488, 518
ilogbd128, 212, 420, 518
ilogbd32, 212, 420, 518
ilogbd64, 212, 420, 518
ilogbf, 211, 415, 518
ilogbl, 211, 415, 518
imaxabs, 185, 413, 518
imaxdiv, 184, 185, 413, 518
isalnum, 166, 167, 168, 411, 511
isalpha, 166, 386, 411, 506, 511
isblank, 166, 411, 506, 511
iscntrl, 166, 167, 168, 411, 511
isdigit, 166, 167, 168, 190, 411, 511
isgraph, 167, 386, 411, 511
islower, 4, 166, 167, 168, 169, 411, 506, 511
isprint, 19, 167, 411, 511
ispunct, 166, 167, 168, 411, 506, 511
isspace, 152, 166, 167, 168, 411, 506, 511
isupper, 166, 168, 169, 411, 506, 511
iswalnum, 387, 388, 389, 434, 511
iswalpha, 386, 387, 389, 434, 506, 511
iswblank, 387, 389, 434, 506, 511
iswcntrl, 387, 389, 434, 511
iswctype, 389, 390, 435, 500, 506, 511
iswdigit, 387, 389, 434, 511
iswgraph, 386, 388, 389, 434, 511
iswlower, 387, 388, 389, 390, 434, 506, 511
iswprint, 386, 388, 389, 434, 511
iswprintf, 386, 387, 388, 389, 434, 506, 511
iswspace, 152, 386, 387, 388, 389, 434, 506, 511
iswupper, 387, 388, 389, 390, 435, 506, 511
iswxdigit, 389, 435, 511
isxdigit, 168, 190, 411, 511
labs, 314, 428, 482, 518
ldexp, 212, 328, 415, 458, 518
ldexp128, 212, 420, 518
ldexpd128, 212, 420, 518
ldexpd64, 212, 420, 518
ldexpf, 212, 415, 518
ldexpl, 212, 415, 518
ldiv, 300, 314, 429, 518
lgamma, 220, 221, 328, 416, 462, 518
lgammad128, 220, 421, 518
lgammad32, 220, 421, 518
lgammad64, 220, 421, 518
lgammaf, 220, 416, 518
lgamma, 220, 416, 518
llabs, 314, 428, 482, 518
lldiv, 300, 314, 429, 518
lllogb, 198, 212, 213, 415, 442, 458, 518
lllogbd128, 212, 420, 518
lllogbd32, 212, 420, 518
lllogbd64, 212, 420, 518
lllogbf, 212, 415, 518
lllogbl, 212, 415, 518
llquantexpd128, 235, 423, 518
llquantexpd32, 235, 423, 518
llquantexpd64, 235, 423, 518
llrint, 222, 223, 328, 417, 442, 447, 463, 464, 484, 489, 518
llrintd128, 222, 421, 518
llrintd32, 222, 421, 518
llrintd64, 222, 421, 518
llrintf, 222, 417, 518
llrintl, 222, 417, 518
llround, 223, 328, 417, 443, 464, 484, 489, 518
llroundd128, 223, 422, 518
llroundd32, 223, 422, 518
llroundd64, 223, 422, 518
llroundf, 223, 417, 518
llroundl, 223, 417, 518
localeconv, 190, 193, 413, 495, 518
localtime, 345, 346, 432, 518
localtime_r, xiii, 346, 432, 572
localtime_s, 432, 518, 552, 553
log, 198, 213, 214, 328, 331, 415, 446, 458, 518
log10, 213, 328, 415, 446, 458, 518
log10d128, 213, 420, 518
log10d32, 213, 420, 518
log10d64, 213, 420, 518
log10f, 213, 415, 518
log10l, 213, 415, 518
log10p1, 213, 214, 415, 446, 458, 518
log10p1d128, 213, 420, 518
log10p1d32, 213, 420, 518
log10p1d64, 213, 420, 518
log10p1f, 213, 415, 518
log10p1l, 213, 415, 518
log1p, 214, 328, 415, 446, 458, 518
log1pd128, 214, 420, 518
log1pd32, 214, 420, 518

Index 605

mbsinit, 381, 382, 433, 519
ISO/IEC 9899:202x (E)  working draft — November 18, 2019  N2455

mbstrwcs, 381, 384, 433, 519, 567
mbstrwcs_s, 434, 519, 567, 568
mbstowcs, 60, 316, 383, 429, 519
mbstowcs_s, 429, 519, 541, 542
mbtowc, 59, 315, 316, 382, 429, 519
memccpy, ii, xi, 319, 429, 511, 572
memchr, 322, 323, 430, 511
memcmp, 39, 256, 321, 429, 511
memcpy, 39, 64, 155, 256, 319, 429, 443, 511
memcpy_s, 430, 512, 543, 544
memmove, 64, 319, 320, 429, 443, 494, 512
memmove_s, 430, 512, 544
memset, 325, 430, 512, 549
memset_s, 430, 512, 549
mkt ime, 343, 344, 431, 519
modf, 215, 216, 327, 328, 416, 459, 483, 519
modfd128, 215, 420, 519
modfd32, 215, 420, 519
modfd64, 215, 420, 519
modff, 215, 416, 483, 519
modfl, 215, 416, 483, 519
mtx_destroy, 335, 431, 512
mtx_init, 333, 335, 336, 431, 512
mtx_lock, 336, 431, 499, 512
mtx_timedlock, 336, 431, 499, 512
mtx_trylock, 336, 337, 431, 512
mtx_unlock, 336, 337, 431, 499, 512
nan, 227, 278, 356, 417, 441, 465, 519
nand128, 227, 422, 519
nand32, 227, 422, 519
nand64, 227, 422, 519
nanf, 227, 417, 519
nanl, 227, 417, 519
nearbyint, 222, 328, 417, 441, 444, 447, 459, 463, 484, 519
nearbyintd128, 222, 421, 519
nearbyintd32, 222, 421, 519
nearbyintd64, 222, 421, 519
nearbyintf, 222, 417, 519
nearbyintl, 222, 417, 519
nextafter, 227, 228, 328, 331, 417, 444, 445, 465, 466, 519
nextafterd128, 227, 422, 519
nextafterd32, 227, 422, 519
nextafterfd64, 227, 422, 519
nextafterf, 227, 417, 519
nextafterl, 227, 417, 519
nextdown, 228, 229, 418, 442, 466, 519
nextdownd128, 229, 422, 519
nextdownd32, 229, 422, 519
nextdownfd64, 229, 422, 519
nextdownf, 228, 418, 519
nextdownl, 229, 418, 519
nexttoward, 228, 328, 331, 418, 444, 445, 466, 519
nexttowardl128, 228, 422, 519
nexttowardl32, 228, 422, 519
nexttowardl64, 228, 422, 519
nexttowardl, 228, 418, 519
nextup, 228, 418, 442, 466, 519
nextupd128, 228, 422, 519
nextupd32, 228, 422, 519
nextupd64, 228, 422, 519
nextupf, 228, 418, 519
nextupl, 228, 418, 519
perror, 299, 427, 519
pow, ii, 217, 218, 328, 331, 416, 446, 460, 519
powd128, 218, 421, 519
powd32, 218, 421, 519
powd64, 218, 331, 421, 519
powf, 217, 416, 519
powl, 217, 416, 444, 519
pown, 218, 416, 446, 461, 519
pownd128, 218, 421, 519
pownd32, 218, 421, 519
pownd64, 218, 421, 519
pownf, 218, 416, 519
pownl, 218, 416, 519
powr, 218, 416, 446, 461, 519
powrd128, 218, 421, 519
powrd32, 218, 421, 519
powrd64, 218, 421, 519
powrf, 218, 416, 519
powrl, 218, 416, 519
printf_s, 428, 519, 529, 530
puts, 145, 268, 295, 427, 519
putwc, 268, 369, 433, 519
putwchar, 268, 369, 433, 519
qsort, 312, 314, 428, 489, 519
qsort_s, 429, 519, 539, 540, 541
quantized128, 234, 423, 519
quantized32, 234, 423, 519
quantized64, 234, 423, 519
quantum128, 235, 423, 519
quantum32, 235, 423, 519
quantum64, 235, 423, 519
quick_exit, 243, 310, 311, 428, 489, 496, 498, 499, 505, 519, 573
raise, 242, 243, 244, 250, 309, 424, 496, 519
rand, 300, 306, 307, 428, 519
realloc, 307, 308, 309, 393, 428, 489, 498, 505, 519
remainder, i, 226, 329, 331, 417, 441, 442, 445, 465, 484, 504, 519
remainderd128, 226, 422, 519
remainderd32, 226, 422, 519
strftime
strfromd64
strfromd128
strfromd
strtod
strspn
strcmp
strrchr
strchr
strcat_s
strcat
sscanf
srand
sqrtl
strtof
strncpy
sqrtd64
sqrtd128
sqrt
strtoimax
strtok
strtok_s
strtok
strtold
Index

609
wcsstr, 378, 433, 514
wcstod, 360, 361, 363, 370, 433, 443, 489, 505, 514
wcstod128, 370, 372, 505, 514
wcstod32, 370, 371, 505, 514
wcstod64, 370, 371, 505, 514
wcstof, 363, 370, 433, 489, 505, 514
wcstoinax, 185, 186, 413, 514
wcstok, 379, 433, 499, 500, 514
wcstok_s, 434, 514, 564, 565
wcstol, 186, 360, 361, 363, 373, 374, 433, 514
wcstold, 360, 361, 363, 373, 374, 433, 489, 505, 514
wcstombs, 317, 383, 429, 514
wcstombs_s, 429, 514, 542, 543
wcstoul, 186, 360, 361, 363, 373, 374, 433, 514
wcstoumax, 185, 186, 413, 514
wcsxfrm, 376, 433, 499, 514
wctob, 381, 386, 433, 521
wctomb, 315, 316, 317, 382, 429, 521
wctomb_s, 429, 521, 541
wctrans, 390, 391, 392, 433, 521
wctype, 389, 390, 384, 399, 433, 521
wmemchr, 379, 380, 433, 521
wmemcmp, 376, 377, 433, 521
wmemcpy, 374, 433, 521
wmemcpy_s, 434, 521, 562
wmemmove, 374, 375, 433, 521
wmemmove_s, 434, 521, 562, 563
wmemset, 380, 433, 521
wprintf, 175, 176, 184, 268, 365, 366, 432, 443, 514
wprintf_s, 434, 521, 559, 560
wscanf, 175, 176, 268, 366, 369, 432, 443, 521
wscanf_s, 434, 521, 559, 560

future directions, 392

identifier
__VA_ARGS__, 139, 140, 145, 514, 574
__func__, 50, 156, 491, 511, 574

identifier prefix
__DECIMAL_DIG, 27, 280, 303, 358, 371, 447, 448
__H__, 153
__STDC_VERSION__, 153
__r, 345
__STDC__, 151
__STDC_WANT_IEC_60559__, 440, 512

ATOMIC__, 393, 508
atomic__, 393, 508
c, 327–329
cnd__, 394, 508
cr, 393
d, 329, 330
d128, 329, 393
d32, 329, 330, 393
d64, 327, 329, 330, 393
DBL__, 392, 508
DEC, 413
DEC128__, 29, 392, 508
DEC32__, 29, 392, 508
DEC64__, 29, 392, 508
DEC, 392, 508
E, 170, 392, 508
FE__, 172, 173, 392, 508
FLT__, 392, 508
FP__, 196, 392, 508
INT, 263, 266, 393, 426, 508
int, 261, 393, 426, 508
INT_FAST, 264, 426
int_fast, 262, 426
INT_LEAST, 263, 426
int_least, 261, 266, 426
is, 392, 394, 508
LC__, 189, 392, 508
LDBL__, 392, 508
llquantexpd, 235, 330, 446, 518
MATH__, 392, 508
mem, 393, 508
memory__, 393
memory_order__, 393
mtx__, 394, 508
PRI, 184, 392, 508
PRId, 184, 413
PRIdFAST, 184, 413
PRIdLEAST, 184, 413
PRIi, 184, 413
PRIiFAST, 184, 413
PRIiLEAST, 184, 413
PRIo, 184, 413
PRIoFAST, 184, 413
PRIoLEAST, 184, 413
PRIx, 184, 413
PRIxFAST, 184, 413
PRIxFAST, 184, 413
PRIxLEAST, 184, 413
quantized, 234, 330, 445, 519
quantumd, 235, 330, 446, 519
samequantumd, 234, 235, 330, 445, 520

610 Index
425, 508

**ATOMIC_CHAR_LOCK_FREE**, 249, 425, 508

**ATOMIC_FLAG_INIT**, 249, 257, 258, 425, 509

**ATOMIC_INT_LOCK_FREE**, 249, 425, 508

**ATOMIC_LLONG_LOCK_FREE**, 249, 425, 509

**ATOMIC_LONG_LOCK_FREE**, 249, 425, 509

**ATOMIC_POINTER_LOCK_FREE**, 249, 425, 509

**ATOMIC_SHORT_LOCK_FREE**, 249, 425, 509

**ATOMIC_WCHAR_T_LOCK_FREE**, 249, 425, 509

**bitand**, 187, 413, 515

**bitor**, 187, 413, 515

**bool**, 515

**BUFSIZ**, 267, 269, 275, 426, 515

**CHAR_BIT**, 21, 23, 39, 91, 413, 438, 515

**CHAR_MAX**, 21, 23, 191, 192, 413, 438, 515

**CHAR_MIN**, 21, 23, 37, 413, 438, 515

**CHAR_WIDTH**, 21, 515

**CLOCKS_PER_SEC**, 342, 343, 431, 515

**CMPLX**, 157, 163, 164, 411, 515

**CMPLXF**, 163, 164, 411, 515

**CMPLXL**, 163, 164, 411, 515

**compl**, 187, 413, 515


**CR_DECIMAL_DIG**, 24, 447, 516

**d32add**, 31, 176, 329, 516

**d32div**, 31, 176, 329, 516

**d32fma**, 31, 176, 329, 516

**d32mul**, 31, 176, 329, 516

**d32sqrt**, 31, 176, 329, 516

**d32sub**, 31, 176, 329, 516

**d64add**, 31, 176, 329, 516

**d64div**, 31, 176, 329, 516

**d64fma**, 31, 176, 329, 516

**d64mul**, 31, 176, 329, 516

**d64sqrt**, 31, 176, 329, 516

**d64sub**, 31, 176, 329, 516

**ddadd**, 329, 516

**DBL_DECIMAL_DIG**, 26, 28, 412, 438, 509

**DBL_DIG**, 26, 28, 412, 438, 509

**DBL_EPSILON**, 27, 28, 412, 439, 483, 509

**DBL_HAS_SUBNORM**, 25, 28, 412, 509

**DBL_MAX**, 27, 28, 412, 439, 483, 509

**DBL_MAX_10_EXP**, 27, 28, 412, 438, 509

**DBL_MAX_EXP**, 27, 28, 412, 438, 483, 509

**DBL_MIN**, 27, 28, 412, 439, 483, 509

**DBL_MIN_10_EXP**, 26, 28, 412, 438, 509

**DBL_MIN_EXP**, 26, 28, 412, 438, 483, 509

**DBL_NORM_MAX**, 27, 439, 509

**DBL_TRUE_MIN**, 27, 28, 412, 509

**dfma**, 329, 516

**DEC128_EPSILON**, 30, 439, 509

**DEC128_MANT_DIG**, 29, 439, 509

**DEC128_MAX**, 30, 439, 509

**DEC128_MAX_EXP**, 29, 439, 509

**DEC128_MIN**, 30, 439, 509

**DEC128_MIN_EXP**, 29, 439, 509

**DEC128_TRUE_MIN**, 30, 439, 509

**DEC64_EPSILON**, 30, 439, 509

**DEC64_MAX**, 30, 439, 509

**DEC64_MAX_EXP**, 29, 439, 509

**DEC64_MIN**, 30, 439, 509

**DEC64_MIN_EXP**, 29, 439, 509

**DEC64_TRUE_MIN**, 30, 439, 509

**DEC_EVAL_METHOD**, 24, 29, 56, 195, 438, 502, 503, 510

**DEC_INFINITY**, 196, 234, 510

**DEC_NAN**, 196, 510

**dfma**, 329, 516

**dmul**, 329, 516

**dsub**, 329, 516

**EDOM**, 198, 411, 510


**EOL**, 510

**ERANGE**, 170, 185, 186, 199, 303, 304, 306, 371, 373, 374, 411, 504, 505, 510

**EXIT_FAILURE**, 300, 310, 428, 510

**EXIT_SUCCESS**, 300, 310, 338, 428, 510

**false**, 235, 259, 393, 426, 452, 582, 517

**FE_ALL_EXCEPT**, 84, 172, 412, 485, 510

**FE_DEC_DOWNWARD**, 172, 176, 412, 446, 510

**FE_DEC_TONEAREST**, 172, 176, 412, 446, 510

**FE_DEC_TONEARESTFROMZERO**, 172, 176, 412, 446, 510

**FE_DEC_TOWARDZERO**, 172, 176, 412, 446, 510

**FE_DEC_UPWARD**, 172, 176, 412, 446, 510
INT16_C, 511
INT16_MAX, 511
INT16_MIN, 511
INT32_C, 511
INT32_MAX, 511
INT32_MIN, 511
INT64_C, 511
INT64_MAX, 511
INT64_MIN, 511
INT8_C, 511
INT8_MAX, 511
INT8_MIN, 511
INT_MAX, 22, 36, 137, 198, 212, 413, 438, 482, 497, 498, 511
INT_MIN, 22, 36, 198, 413, 438, 482, 511
INT_WIDTH, 22, 225, 511
INTMAX_C, 266, 426, 511
INTMAX_MAX, 185, 186, 264, 426, 511
INTMAX_MIN, 185, 186, 264, 426, 511
INTMAX_WIDTH, 265, 426, 511
INTPTR_MAX, 264, 426, 511
INTPTR_MIN, 264, 426, 511
INTPTR_WIDTH, 264, 426, 511
iscanonical, 24, 200, 414, 444, 511
isegsig, 239, 418, 443, 470, 511
isfinite, 200, 414, 443, 444, 473, 474, 511
isgreater, 237, 238, 418, 443, 467, 511
isgreaterequal, 238, 418, 443, 452, 466, 511
isinf, 200, 201, 414, 444, 459, 473, 474, 511
isless, 238, 418, 443, 452, 511
islessequal, 238, 418, 443, 511
islessgreater, 238, 239, 418, 444, 511
isnan, 201, 414, 444, 466, 473, 474, 511
isnormal, 201, 414, 443, 444, 511
issignaling, 201, 202, 414, 443, 444, 511
issubnormal, 202, 414, 444, 511
isunordered, 239, 418, 443, 444, 511
izero, 202, 414, 444, 511
kill_dependency, 15, 252, 253, 425, 518
L_tmpnam, 267, 272, 426, 519
L_tmpnam_s, 427, 519, 524, 525
LC_ALL, 189, 190, 193, 413, 511
LC_COLLATE, 189, 190, 322, 376, 413, 511
LC_CTYPE, 189, 190, 300, 315, 316, 381, 386, 389–391, 413, 499, 500, 511, 541
LC_MONETARY, 189, 190, 193, 413, 511
LC_NUMERIC, 189, 190, 193, 413, 511
LC_TIME, 189, 190, 345, 347, 413, 511
LDBL_DECIMAL_DIG, 26, 392, 412, 439, 511
LDBL_DIG, 26, 412, 439, 511
LDBL_EPSILON, 27, 412, 439, 483, 511
LDBL_HAS_SUBNORM, 25, 412, 511
LDBL_MANT_DIG, 25, 412, 439, 483, 511
LDBL_MAX, 27, 412, 439, 483, 511
LDBL_MAX_10_EXP, 27, 412, 439, 511
LDBL_MAX_EXP, 26, 412, 439, 483, 511
LDBL_MIN, 27, 412, 439, 483, 511
LDBL_MIN_10_EXP, 26, 412, 439, 511
LDBL_MIN_EXP, 26, 412, 439, 483, 511
LDBL_NORMAL_MAX, 27, 439, 511
LDBL_TRUE_MIN, 27, 412, 511
LONGLONG, 23, 306, 374, 413, 438, 482, 518
LONGLONG_MIN, 23, 235, 306, 374, 413, 438, 482, 518
LONGLONG_WIDTH, 23, 518
llquantexp, i, 330, 518
LONG_MAX, 22, 198, 212, 306, 374, 413, 438, 482, 518
LONG_MIN, 22, 198, 306, 374, 413, 438, 482, 518
LONG_WIDTH, 22, 518
MATH_ERR except, 198, 199, 414, 453, 504, 511
math_errnohandling, 153, 198, 199, 414, 453, 488, 495, 504, 519, 574
MATH_ERRNO, 198, 199, 414, 504, 511
MB_CUR_MAX, 152, 300, 316, 351, 352, 383, 428, 519, 541, 566
MB_LEN_MAX, 21, 152, 300, 413, 438, 519
NAN, 196, 278, 302–304, 356, 370–372, 413, 441
NDEBUG, 120, 153, 156, 410, 519
noreturn, 318, 429, 519
not, 187, 413, 519
not_eq, 187, 413, 519
NULL, 46, 189, 227, 260, 267, 300, 301, 319, 322, 325, 342, 350, 351, 353, 376, 379, 382, 413, 426, 428, 429, 431, 432, 504, 519, 548, 549, 565
offsetof, i, ii, 93, 260, 426, 496, 519
ONCE_FLAG_INIT, 332, 431, 519
or, 187, 413, 519
or_eq, 187, 413, 519
PRICE_MAX, 184
PRICPTR, 184
PRID32, 512
PRID64, 512
PRIDFAST32, 184, 512
PRIDFAST64, 512
PRIDLEAST32, 512
PRIDLEAST64, 512
PRIDMAX, 184, 413, 512
PRIDPTR, 184, 413, 512
PRIi32, 512
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIi64</td>
<td>512</td>
</tr>
<tr>
<td>PRIiFAST32</td>
<td>512</td>
</tr>
<tr>
<td>PRIiFAST64</td>
<td>512</td>
</tr>
<tr>
<td>PRIileast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIileast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIimax</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIiptr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIo32</td>
<td>512</td>
</tr>
<tr>
<td>PRIo64</td>
<td>512</td>
</tr>
<tr>
<td>PRIofast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIofast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIoleast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIoleast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIOMUX</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIiptr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIu32</td>
<td>512</td>
</tr>
<tr>
<td>PRIu64</td>
<td>512</td>
</tr>
<tr>
<td>PRIufast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIufast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIuleast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIuleast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIumax</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIuptr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIX32</td>
<td>512</td>
</tr>
<tr>
<td>PRIX64</td>
<td>512</td>
</tr>
<tr>
<td>PRIxFast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIxFast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIxLeast32</td>
<td>512</td>
</tr>
<tr>
<td>PRIxLeast64</td>
<td>512</td>
</tr>
<tr>
<td>PRIxMax</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIxPTr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PRIxptr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>PTRDIFF_MAX</td>
<td>265, 426, 519</td>
</tr>
<tr>
<td>PTRDIFF_MIN</td>
<td>265, 426, 519</td>
</tr>
<tr>
<td>PTRDIFF_WIDTH</td>
<td>265, 519</td>
</tr>
<tr>
<td>putc</td>
<td>268, 294, 427, 519</td>
</tr>
<tr>
<td>putchar</td>
<td>268, 294, 427, 519</td>
</tr>
<tr>
<td>quantize</td>
<td>31, 176, 330, 519</td>
</tr>
<tr>
<td>quantum</td>
<td>31, 330, 519</td>
</tr>
<tr>
<td>RAND_MAX</td>
<td>300, 306, 307, 428, 519</td>
</tr>
<tr>
<td>RSIZE_MAX</td>
<td>426, 520, 524, 525, 526, 530, 531, 534–536, 538, 540–549, 551, 552, 554, 555, 557, 558, 561–564, 566–568</td>
</tr>
<tr>
<td>samequantum</td>
<td>330, 520</td>
</tr>
<tr>
<td>SCNUMAX</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>SCNuptr</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>SCNXMAX</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>SCNXPTR</td>
<td>184, 413, 512</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>268, 297, 426, 520</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>268, 270, 297, 426, 520</td>
</tr>
<tr>
<td>SEEK_SET</td>
<td>268, 297, 298, 426, 498, 520</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>22, 413, 438, 520</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>21, 413, 438, 520</td>
</tr>
<tr>
<td>SHRT_WIDTH</td>
<td>22, 520</td>
</tr>
<tr>
<td>SIG_ATOMIC_MAX</td>
<td>265, 266, 426, 512</td>
</tr>
<tr>
<td>SIG_ATOMIC_MIN</td>
<td>265, 266, 426, 512</td>
</tr>
<tr>
<td>SIG_ATOMIC_WIDTH</td>
<td>265, 426, 512</td>
</tr>
<tr>
<td>SIG_DFL</td>
<td>242, 243, 424, 504, 512</td>
</tr>
<tr>
<td>SIG_ERR</td>
<td>242, 243, 424, 496, 512</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>242, 243, 424, 500, 512</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>242, 309, 424, 512</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>198, 242, 243, 424, 485, 496, 500, 508, 512</td>
</tr>
<tr>
<td>SIGILL</td>
<td>242, 243, 424, 496, 500, 512</td>
</tr>
<tr>
<td>SIGINT</td>
<td>242, 424, 512</td>
</tr>
<tr>
<td>signbit</td>
<td>201, 414, 443, 465, 520</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>242, 243, 424, 496, 500, 512</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>242, 424, 512</td>
</tr>
<tr>
<td>SIZE_MAX</td>
<td>265, 426, 520, 524</td>
</tr>
<tr>
<td>SIZE_WIDTH</td>
<td>265, 426, 520</td>
</tr>
<tr>
<td>SNAN</td>
<td>196, 441, 520</td>
</tr>
<tr>
<td>SNAND128</td>
<td>196, 520</td>
</tr>
<tr>
<td>SNAND32</td>
<td>196, 520</td>
</tr>
<tr>
<td>SNAND64</td>
<td>196, 520</td>
</tr>
<tr>
<td>SNANF</td>
<td>196, 441, 520</td>
</tr>
<tr>
<td>SNANL</td>
<td>196, 441, 520</td>
</tr>
<tr>
<td>static_assert</td>
<td>ii, 156, 410, 520</td>
</tr>
<tr>
<td>thread_local</td>
<td>332, 431, 520</td>
</tr>
<tr>
<td>TIME_UTC</td>
<td>335, 336, 339, 342, 344, 431, 505, 513</td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>268, 272, 426, 520</td>
</tr>
<tr>
<td>TMP_MAX_S</td>
<td>427, 520, 524, 525, 526</td>
</tr>
<tr>
<td>true</td>
<td>235, 259, 393, 426, 452, 482, 521</td>
</tr>
<tr>
<td>TSS_DTOR_ITERATIONS</td>
<td>332, 338, 431, 521</td>
</tr>
<tr>
<td>uchar</td>
<td>21, 23, 413, 438, 521</td>
</tr>
<tr>
<td>uchar_width</td>
<td>21, 521</td>
</tr>
<tr>
<td>UINT16_C</td>
<td>513</td>
</tr>
<tr>
<td>UINT16_MAX</td>
<td>513</td>
</tr>
<tr>
<td>UINT32_C</td>
<td>513</td>
</tr>
<tr>
<td>UINT32_MAX</td>
<td>513</td>
</tr>
<tr>
<td>UINT64_C</td>
<td>266, 513</td>
</tr>
<tr>
<td>UINT64_MAX</td>
<td>513</td>
</tr>
<tr>
<td>UINT8_C</td>
<td>513</td>
</tr>
<tr>
<td>UINT8_MAX</td>
<td>513</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>22, 137, 413, 438, 482, 514</td>
</tr>
<tr>
<td>UINT_WIDTH</td>
<td>22, 514</td>
</tr>
<tr>
<td>UINTMAX_C</td>
<td>266, 426, 514</td>
</tr>
<tr>
<td>UINTMAX_MAX</td>
<td>184–186, 265, 426, 447, 514</td>
</tr>
</tbody>
</table>
obsolete

__STDC_IEC_559_COMPLEX__, 23, 149, 151, 471, 512
__STDC_IEC_559__, 23, 149, 151, 424, 440, 512

ATOMIC_VAR_INIT, 250, 393, 425, 509

DECIMAL_DIG, i, xiii, 24, 26–28, 280, 303, 358, 371, 392, 412, 438, 447, 448, 516, 572

gets, 518, 536, 573

stream

stderr, 145, 268, 269, 270, 274, 290, 364, 426, 507, 520

structure member

currency_symbol, 189, 191, 193, 516
decimal_point, 189, 191, 516
frac_digits, 189, 191, 193, 517
grouping, 189, 191, 192, 518
int_curr_symbol, 189, 192, 193, 518
int_frac_digits, 189, 192, 193, 518
int_n_cs_precedes, 189, 192, 193, 518
int_n_sep_by_space, 189, 192, 193, 518
int_n_sign_posn, 189, 192, 193, 518
int_p_cs_precedes, 189, 192, 193, 518
int_p_sep_by_space, 189, 192, 193, 518
int_p_sign_posn, 189, 192, 193, 518
mon_decimal_point, 189, 191, 193, 519
mon_grouping, 189, 191–193, 519
mon_thousands_sep, 189, 191, 193, 519
n_cs_precedes, 189, 191, 193, 519
n_sep_by_space, 189, 191–193, 519
n_sign_posn, 189, 192, 193, 519
negative_sign, 189, 191–193, 519
p_cs_precedes, 189, 191, 193, 519
p_sep_by_space, 189, 191–194, 519
p_sign_posn, 189, 191, 194, 519
positive_sign, 189, 191–193, 519
thousands_sep, 189, 191, 520
tm_hour, 342, 344, 345, 347, 348, 520, 551
tm_isdst, 343, 344, 348, 520
tm_mday, 342, 343–345, 347, 520, 551
tm_min, 342, 344, 345, 347, 348, 520, 551
tm_mon, 342, 343–345, 347, 520, 551
tm_sec, 342, 344, 345, 347, 348, 521, 551
tm_wday, 343, 344, 345, 347, 348, 521, 551
tm_year, 343, 347, 348, 521
tv_nsec, 343, 344, 345, 347, 348, 521, 551
tv_sec, 342, 344, 521
tv_usec, 342, 344, 521

summary, 410
terms, 152
type

 Void type, 43, 89
 Complex types, 89, 157
 _Decimal32_t, 195, 503, 510
 _Decimal64_t, 195, 503, 510
 Imaginary types, 157
 atomic_bool, 254, 257, 425, 508
 atomic_char, 254, 425, 508
 atomic_char16_t, 254, 425, 508
 atomic_char32_t, 254, 425, 508
 atomic_flag, 249, 257, 258, 425, 426, 509
 atomic_int, 250, 254, 425, 509
 atomic_int_fast16_t, 255, 425, 509
 atomic_int_fast32_t, 255, 425, 509
 atomic_int_fast64_t, 255, 425, 509
 atomic_int8_t, 254, 255, 425, 509
 atomic_intmax_t, 255, 425, 509
 atomicintptr_t, 255, 425, 509
 atomic_llong, 254, 425, 509
atomic_long, 254, 425, 509
atomic_ptrdiff_t, 255, 425, 509
atomic_schar, 254, 425, 509
atomic_short, 254, 425, 509
atomic_size_t, 255, 425, 509
atomic_uchar, 254, 425, 509
atomic_uint, 254, 425, 509
atomic_uint_fast16_t, 255, 425, 509
atomic_uint_fast32_t, 255, 425, 509
atomic_uint_fast64_t, 255, 425, 509
atomic_uint_fast8_t, 255, 425, 509
atomic_uint_least16_t, 255, 425, 509
atomic_uint_least32_t, 255, 425, 509
atomic_uint_least64_t, 255, 425, 509
atomic_uintmax_t, 255, 425, 509
atomic_long, 254, 425, 509
cchar, 254, 425, 509
cchar16_t, i, 58, 60, 113, 148, 254, 350, 351, 432, 501, 515
cchar32_t, 58, 60, 113, 148, 254, 350, 351, 352, 432, 501, 515
clock_t, 342, 343, 431, 505, 515
cnd_t, 332, 333–335, 431, 509
constraint_handler_t, 429, 515, 537
div_t, 116, 300, 314, 428, 516
double, 89
double_t, 195, 413, 450, 503, 507, 516
enum, 89
femode_t, 171, 172, 179, 180, 412, 517
fenv_t, 84, 171, 173, 181, 182, 412, 464, 517
fexcept_t, 171, 177, 178, 412, 495, 517
float, 89
float_t, 195, 413, 450, 503, 507, 517
fpos_t, 267, 269, 296, 297, 426, 427, 517
imaxdiv_t, 184, 185, 413, 518
int, 43, 53, 89
int16_t, 511
int32_t, 511
int64_t, 511
int8_t, 261, 511
int_fast16_t, 255, 262, 511
int_fast32_t, 184, 255, 262, 511
int_fast64_t, 255, 262, 511
int_fast8_t, 255, 262, 511
int_least16_t, 254, 262, 511
int_least32_t, 255, 261, 262, 511
int_least64_t, 255, 262, 511
int_least8_t, 254, 262, 511
intmax_t, 137, 185, 186, 216–219, 224, 225, 255, 262, 266, 277, 284, 355, 360, 413, 416, 417, 421, 422, 426, 511, 573
intptr_t, 255, 262, 426, 511
jmp_buf, 240, 241, 424, 518, 571
ldiv_t, 300, 314, 428, 429, 518
ldiv_t, 300, 314, 428, 429, 518
max_align_t, 41, 42, 260, 426, 519
memory_order, 249, 250, 253, 255–258, 393, 425, 426, 512
mtx_t, 332, 334–337, 431, 512
once_flag, 332, 333, 431, 519
ptrdiff_t, 76, 77, 249, 255, 260, 265, 277, 284, 355, 360, 426, 492, 519
sig_atomic_t, 12, 242, 243, 265, 266, 424, 487, 496, 520
signed, 89
thrd_start_t, 332, 337, 431, 513
thrd_t, 332, 337, 338, 431, 513
time_t, 342, 343, 344, 346, 431, 432, 505, 520, 552
tss_dtor_t, 332, 339, 431, 513
tss_t, 332, 339, 430, 431, 513
uint16_t, 513
uint32_t, 513
uint64_t, 225, 513
uint8_t, 513
uint_fast16_t, 255, 262, 513
uint_fast32_t, 255, 262, 513
uint_fast64_t, 255, 262, 513
uint_fast8_t, 255, 262, 513
uint_least16_t, 255, 261, 262, 350, 513
uint_least32_t, 255, 262, 350, 513
uint_least64_t, 255, 262, 66, 513
uint_least8_t, 254, 262, 513
uintmax_t, 137, 184–186, 224, 225, 255, 262, 266, 277, 284, 355, 360, 413, 417, 422, 426, 514, 573
uintptr_t, 255, 262, 426, 514
unsigned, 89, 277, 284, 355, 360
void, 46, 89
wchar_t, 386, 390, 391, 434, 435, 521
wchar_t, 386, 389, 390, 434, 435, 521
wint_t, 265, 266, 277, 279, 281, 353, 355, 357, 367–369, 381, 386–390, 432–435, 500, 521
use of functions, 154
lifetime, 35
limits
environmental, see environmental limits
implementation, see implementation limits
numerical, see numerical limits
translation, see translation limits
line buffered, 270
line buffered stream, 270
line number, 145, 148
line preprocessing directive, 145
lines, 9, 268
preprocessing directive, 136
linkage, 34, 87, 102, 106, 131, 133, 151
llabs function, 314, 428, 482, 518
lldiv function, 300, 314, 429, 518
lldiv_t type, 300, 314, 428, 429, 518
llogb function, 198, 212, 213, 415, 442, 458, 518
llogb type-generic macro, 329
llogbd128 function, 212, 420, 518
llogbd32 function, 212, 420, 518
llogbd64 function, 212, 420, 518
llogbf function, 212, 415, 518
llogbl function, 212, 415, 518
LONGLONG_MAX macro, 23, 306, 374, 413, 438, 482, 518
LONGLONG_MAX macro, 23, 235, 306, 374, 413, 438, 482, 518
LLONG_WIDTH macro, 23, 518
llquantexp macro, i, 330, 518
llquantexp identifier prefix, 235, 330, 446, 518
llquantexpd128 function, 235, 423, 518
llquantexpd32 function, 235, 423, 518
llquantexpd64 function, 235, 423, 518
llrint function, 222, 223, 328, 417, 442, 447, 463, 464, 484, 489, 518
llrint type-generic macro, 329
llrintd128 function, 222, 421, 518
llrintd32 function, 222, 421, 518
llrintd64 function, 222, 421, 518
llrintf function, 222, 417, 518
llrintl function, 222, 417, 518
llround function, 223, 328, 417, 443, 464, 484, 489, 518
llround functions, 223
llround type-generic macro, 329
llroundd128 function, 223, 422, 518
llroundd32 function, 223, 422, 518
llroundd64 function, 223, 422, 518
llroundf function, 223, 417, 518
llroundl function, 223, 417, 518
local time, 342
locale, 3
locale-specific behavior, 3, 506
localeconv function, 190, 193, 413, 495, 518
localization header, 189, 392
localtime function, 345, 346, 432, 518
localtime_r function, xii, 346, 432, 572
localtime_s function, 432, 518, 552, 553
log function, 198, 213, 214, 328, 331, 415, 446, 458, 518
log type-generic macro, 328
log10 function, 213, 328, 415, 446, 458, 518
log10 type-generic macro, 329
log10d128 function, 213, 420, 518
log10d32 function, 213, 420, 518
log10d64 function, 213, 420, 518
log10f function, 213, 415, 518
log10l function, 213, 415, 518
log10pl function, 213, 415, 446, 458, 518
log10pl type-generic macro, 329
log10pld128 function, 213, 420, 518
log10pld32 function, 213, 420, 518
log10pld64 function, 213, 420, 518
log10plf function, 213, 415, 518
log10pll function, 213, 415, 518
log10pl function, 214, 328, 415, 446, 458, 518
log10pl type-generic macro, 329
log10pld128 function, 214, 420, 518
log10pld32 function, 214, 420, 518
log10pld64 function, 214, 420, 518
log10plf function, 214, 415, 518
log10pll function, 214, 415, 518
log2 function, 214, 328, 415, 446, 458, 518
log2 type-generic macro, 329
log2d128 function, 214, 420, 518
log2d32 function, 214, 420, 518
log2d64 function, 214, 420, 518
log2f function, 214, 416, 518
log2l function, 214, 416, 518
log2pl function, 214, 415, 446, 458, 518
log2pl type-generic macro, 329
**log2p1d128** function, 215, 420, 518
**log2p1d32** function, 215, 420, 518
**log2p1d64** function, 215, 420, 518
**log2pf** function, 215, 416, 518
**log2pl** function, 215, 416, 518
logarithmic functions
  - complex, 161, 479
  - real, 209, 456
**logb** function, 212, 213, 215, 328, 416, 442, 445, 457, 459, 473, 483, 518
**logb** type-generic macro, 329
**logbd128** function, 215, 420, 518
**logbd32** function, 215, 420, 518
**logbd64** function, 215, 420, 518
**logbf** function, 215, 416, 483, 518
**logbl** function, 215, 416, 483, 518
**logd128** function, 213, 420, 518
**logd32** function, 213, 420, 518
**logd64** function, 213, 420, 518
**logf** function, 213, 415, 518
logarithmic operators
  - AND (&&), 15, 80
  - negation (!), 73
  - OR (||), 80
  - OR (||), 15
logical source lines, 9
**logl** function, 213, 415, 518
**logpl** function, 214, 415, 446, 458, 518
**logpl** type-generic macro, 329
**logpld128** function, 214, 420, 518
**logpld32** function, 214, 420, 518
**logpld64** function, 214, 420, 518
**logplf** function, 214, 415, 518
**logpl** function, 214, 415, 518
**long double _Complex** type, 37
**long double _Complex** type conversion, 44
**long double _Imaginary** type, 471
**long double** type suffix, `l` or `L`, 55
**long double** type, 36, 89
**long double** type conversion, 44
**long int** type, 36, 89
**long int** type conversion, 43, 44
**long int** type suffix, `l` or `L`, 53
**long long int** type, 36, 89
**long long int** type conversion, 43, 44
**long long integer suffix, ll or LL, 53**
**LONG_MAX** macro, 22, 198, 212, 306, 374, 413, 438, 482, 518
**LONG_MIN** macro, 22, 198, 306, 374, 413, 438, 482, 518
**LONG_WIDTH** macro, 22, 518
**longjmp** function, 240, 241, 310, 311, 424, 495, 498, 518, 571
loop body, 127
low-order bit, 4
lowercase letters, 17
**lrint** function, 222, 223, 328, 417, 442, 447, 463, 464, 484, 489, 518
**lrint** type-generic macro, 329
**lrintd128** function, 222, 421, 518
**lrintd32** function, 222, 421, 518
**lrintd64** function, 222, 421, 519
**lrintf** function, 222, 417, 519
**lrintl** function, 222, 417, 519
**lround** function, 223, 328, 417, 443, 464, 484, 489, 519
**lround** functions, 223
**lround** type-generic macro, 329
**lroundd128** function, 223, 422, 519
**lroundd32** function, 223, 422, 519
**lroundd64** function, 223, 422, 519
**lroundf** function, 223, 417, 519
**lroundl** function, 223, 417, 519
lvalue, 45, 65, 70, 72, 82, 97
lvalue conversion, 46, 82, 83
macro argument substitution, 140
macro definition
  - library function, 154
macro invocation, 139
macro name, 139
  - length, 20
  - predefined, 147, 151
  - redefinition, 139
scope, 142
macro parameter, 139
macro preprocessor, 135
macro replacement, 139
magnitude, complex, 162
manipulation functions
  - complex, 163
  - real, 226, 465
matching failure, 365, 366, 557–559
math rounding direction macros, 197
**MATH** identifier prefix, 392, 508
**MATH_ERREXCEPT** macro, 198, 199, 414, 453, 504, 511
**math_errhandling** macro, 153, 198, 199, 414, 504, 511
mathematics header, 195, 392
**MAX** identifier suffix, 117
**max_align_t** type, 41, 42, 260, 426, 519
maximal munch, 48
maximum functions, 229, 466
maybe_unused attribute, ii, xiv, 118, 120, 519, 572
**MB_CUR_MAX** macro, 152, 300, 316, 351, 352, 383, 428, 519, 541, 566
**MB_LEN_MAX** macro, 21, 152, 300, 413, 438, 519
**mblen** function, ii, 315, 382, 429, 519
**mbrlen** function, 382, 433, 519
**mbtoc16** function, 59, 60, 350, 432, 519
modf function, 215, 420, 519
modff function, 215, 416, 483, 519
modfl function, 215, 416, 483, 519
modf1 function, 215, 416, 483, 519
modifiable lvalue, 46
modification order, 15
modulus functions, 215
modulus, complex, 162
mon_decimal_point structure member, 189, 191, 193, 519
mon_grouping structure member, 189, 191–193, 519
mon_thousands_sep structure member, 189, 191, 193, 519
mtx_ identifier prefix, 394, 508
mtx_destroy function, 335, 431, 512
mtx_init function, 333, 335, 336, 431, 512
mtx_lock function, 336, 431, 499, 512
mtx_plain constant, 332, 336, 431, 512
mtx_recursive constant, 333, 336, 431, 512
mtx_t type, 332, 334–337, 431, 512
mtx_timed constant, 333, 336, 431, 512
mtx_timedlock function, 336, 431, 499, 512
mtx_trylock function, 336, 337, 431, 512
mtx_unlock function, 336, 337, 431, 499, 512
multibyte character, 4, 18, 57
multibyte conversion functions
  wide character, 315, 541
    extended, 381, 566
    restartable, 350, 382, 566
  wide string, 316, 541
    restartable, 383, 567
multibyte string, 152
multibyte/wide character conversion functions, 315, 541
  extended, 381, 566
  restartable, 350, 382, 566
multibyte/wide string conversion functions, 316, 541
  restartable, 383, 567
multidimensional array, 67
multiplication assignment operator (*=), 83
multiplication operator (*), 40, 75, 472
multiplicative expressions, 75, 472
multiply and round to narrower type, 232
n-char sequence, 302
n-wchar sequence, 370
n_cs_precedes structure member, 189, 191, 193, 519
n_sep_by_space structure member, 189, 191–193, 519
n_sign_posn structure member, 189, 192, 193, 519
name
  external, 20, 50, 151
  file, 270
  internal, 20, 50
nearbyint, 34
structure/union member, 34
name spaces, 34
named label, 123
NaN, 24, 441
nan function, 227, 278, 356, 417, 441, 465, 519
NAN macro, 196, 278, 302–304, 356, 370–372, 413, 441
nand128 function, 227, 422, 519
nand32 function, 227, 422, 519
nand64 function, 227, 422, 519
nanf function, 227, 417, 519
nanl function, 227, 417, 519
NDEBUG macro, 120, 153, 156, 410, 519
nearbyint function, 222, 328, 417, 444, 447, 459, 463, 484, 519
nearbyint type-generic macro, 329
nearbyint128 function, 222, 421, 519
nearbyint32 function, 222, 421, 519
nearbyint64 function, 222, 421, 519
nearbyintf function, 222, 417, 519
nearbyintl function, 222, 417, 519
nearest integer functions, 221, 462
negation operator (!), 73
negative zero, 24, 40, 227
negative_sign structure member, 189, 191–193, 519
new line, 19
new-line character, 9, 17, 48, 136, 145
new-line escape sequence (\n), 19, 58, 168
nextafter function, 227, 228, 328, 331, 417, 444, 445, 465, 466, 519
nextafter type-generic macro, 329
nextafter128 function, 227, 422, 519
nextafter32 function, 227, 422, 519
nextafter64 function, 227, 422, 519
nextafterf function, 227, 417, 519
nextafterl function, 227, 417, 519
nextdown function, 228, 229, 418, 442, 466, 519
nextdown type-generic macro, 329
nextdown128 function, 229, 422, 519
nextdown32 function, 229, 422, 519
nextdown64 function, 229, 422, 519
nextdownf function, 228, 418, 519
nextdownl function, 229, 418, 519
nexttoward function, 228, 328, 331, 418, 444, 445, 466, 519
nexttoward type-generic macro, 329
nexttoward128 function, 228, 422, 519
nexttoward32 function, 228, 422, 519
nexttoward64 function, 228, 422, 519
nexttowardf function, 228, 331, 418, 519
nexttowardl function, 228, 418, 519
nextup function, 228, 418, 442, 466, 519
nextup type-generic macro, 329
nextupd128 function, 228, 422, 519
nextupd32 function, 228, 422, 519
nextupd64 function, 228, 422, 519
nextupf function, 228, 418, 519
nextupl function, 228, 418, 519
no linkage, 34
no-return function, 102
nodiscard attribute, ii, xiv, 118, 119, 120, 519, 572
non-canonical, 24
non-canonical representation, 24
non-stop floating-point control mode, 181
nongraphic characters, 19, 58
nonlocal jumps header, 240
noreturn macro, 318, 429, 519
norm, complex, 162
normalized, 550
normalized broken-down time, 550, 551
normalized floating-point numbers, 23
not macro, 187, 413, 519
not-equal-to operator, see inequality operator
not_eq macro, 187, 413, 519
null character (\0), 17, 59, 60
padding of binary stream, 269
nullptr, 46
null pointer constant, 46
null preprocessing directive, 147
null statement, 124
null wide character, 152
number classification macros, 196, 200
numeric conversion functions, 185, 300
wide string, 185, 370
numerical limits, 20
O format modifier, 348
object, 6
object representation, 39
object type, 36
object types, 36
object-like macro, 139
observable behavior, 12
obsolescence, xv, 151, 392
octal constant, 52
octal digit, 52, 58
octal-character escape sequence (\octal_digits), 58
OFF pragma, 146, 408, 473, 519
offsetof macro, i, ii, 93, 260, 426, 496, 519
ON pragma, 84, 146, 174, 179, 180, 182, 408, 449, 450, 459, 463–465, 519
on-off switch, 146
once_flag type, 332, 333, 431, 519
ONCE_FLAG_INIT macro, 332, 431, 519
ones’ complement, 40

Index 621
**opening, 269**
**operand, 61, 64**
**operating system, 10, 312**
**operations on files, 271, 525**
**operator, 61**
**operators, 64**

- `__alignof` operator, 74
- additive, 40, 76
- assignment, 82
- associativity, 64
- equality, 78
- multiplicative, 40, 75, 472
- postfix, 66
- precedence, 64
- preprocessing, 136, 140, 149
- relational, 77
- shift, 77
- `sizeof` operator, 74
- unary, 72
- unary arithmetic, 73

**optional features, see conditional features**
- `or` macro, 187, 413, 519

**OR operators**
- bitwise exclusive (`^`), 40, 80
- bitwise exclusive assignment (`^=`), 83
- bitwise inclusive (`,`), 40, 80
- bitwise inclusive assignment (`|=`), 83
- logical (`|`), 80
- logical (`&`), 15

- `or` macro, 187, 413, 519

**order of evaluation, 64, 82, 140, 141, see also sequence points**

**ordinary identifier name space, 34**

**orientation, 269**

**orientation of stream, 269, 368**

**out-of-bounds store, 570**

**outer scope, 33**

**over-aligned, 42**

- `p_c_s_precedes` structure member, 189, 191, 193, 194, 519
- `p_sep_by_space` structure member, 189, 191–194, 519
- `p_sign_posn` structure member, 189, 191, 193, 519

**padding**
- binary stream, 269
- bits, 39, 261
- structure/union, 39, 92

**parameter, 6**
- array, 132
- ellipsis, 108, 139
- function, 67, 88, 132
- macro, 139
- `main` function, 11
- program, 11

**parameter type list, 108**

**parentheses punctuator (`,`), 108, 125, 127**

**parenthesized expression, 65**

**parse state, 269**

**perform a trap, 7**

**permitted form of initializer, 85**

- `perror` function, 299, 427, 519

- **phase angle, complex, 163**
- physical source lines, 9
- placemarker, 141
- plus operator, unary, 73
- pointer arithmetic, 76
- pointer comparison, 78
- pointer declarator, 105
- pointer operator (`->`), 69
- pointer to a string, 152
- pointer to a wide string, 152
- pointer to function, 67
- pointer type, 38
- pointer type conversion, 46
- pointer, null, 46
- pole error, 198, 208, 213–215, 217–221
- portability, 8, 487
- position indicator, file, *see* file position indicator
- positive difference, 230
- positive difference functions, 229, 466
- `positive_sign` structure member, 189, 191–193, 519
- postfix decrement operator (`--`), 46, 70
- postfix expressions, 66
- postfix increment operator (`,`), 46, 70
- pow function, ii, 217, 218, 328, 331, 416, 446, 460, 519
- `pow` type-generic macro, 328
- `pow128` function, 218, 421, 519
- `pow32` function, 218, 421, 519
- `pow64` function, 218, 331, 421, 519
- power functions
  - complex, 162, 480
  - real, 216, 459
- `powf` function, 217, 416, 519
- `powl` function, 217, 416, 444, 519
- `pown` function, 218, 416, 446, 461, 519
- `pown` type-generic macro, 329
- `pownd128` function, 218, 421, 519
- `pownd32` function, 218, 421, 519
- `pownd64` function, 218, 421, 519
- `pownf` function, 218, 416, 519
- `pown1` function, 218, 416, 519
- `powr` function, 218, 416, 446, 461, 519
- `powrd128` function, 218, 421, 519
- `powrd32` function, 218, 421, 519
- `powrd64` function, 218, 421, 519
- `powrf` function, 218, 416, 519
powr function, 218, 416, 519
pp-number, 62
pragma
  CX_LIMITED_RANGE, viii, 146, 147, 158, 408, 410, 472, 495, 516
  DEFAULT, 146, 408, 516
  FE_DEC_DOWNWARD, 146
  FE_DEC_DYNAMIC, 146, 176, 510
  FE_DEC_TONEAREST, 146
  FE_DEC_TONEARESTFROMZERO, 146
  FE_DEC_UPWARD, 146
  FE_DOWNWARD, 146
  FE_DYNAMIC, 146, 149, 174, 175, 412, 510
  FE_TONEAREST, 146
  FE_TONEARESTFROMZERO, 146
  FE_TOWARDZERO, 146
  FE_UPWARD, 146
  FENV_ACCESS, viii, 84, 146, 147, 173, 174, 179, 180, 182, 408, 412, 448–453, 459, 463–465, 488, 495, 502, 517
  FENV_DEC_ROUND, viii, 56, 146, 147, 172, 175, 176, 408, 412, 517
  FENV_ROUND, viii, 146, 147, 173, 174, 175, 176, 408, 412, 445, 517
  FP_CONTRACT, viii, 65, 146, 147, 179, 180, 408, 414, 473, 495, 502, 510
  OFF, 146, 408, 473, 519
  ON, 84, 146, 174, 179, 180, 182, 408, 449, 450, 459, 463–465, 519
  STDC, 84, 146, 151, 158, 173–175, 179, 180, 182, 199, 408, 410, 412, 414, 449, 450, 459, 463–465, 473, 494, 503, 520
pragma operator, 149
precedence of operators, 64
precedence of syntax rules, 9
precision, 40, 42, 276, 354
  excess, 25, 45, 130
predefined macro names, 147, 151
preferred quantum exponent, 30
prefix decrement operator (--), 46, 73
prefix increment operator (++), 46, 73
preprocessing, 136
preprocessing concatenation, 140
preprocessing directive, 136
preprocessing directives, 9, 135
preprocessing file, 9, 135
preprocessing files, 9
preprocessing numbers, 48, 62
preprocessing operators
  #, 140

Pragma operator, 149
preprocessing tokens, 9, 48, 136
preprocessing translation unit, 9
preprocessor, 135
PRI identifier prefix, 184, 392, 508
PRIcFASTN macros, 184
PRIcLEASTN macros, 184
PRIcMAX macro, 184
PRIcN macros, 184
PRIcPTR macro, 184
PRIid identifier prefix, 184, 413
PRIid32 macro, 512
PRIid64 macro, 512
PRIidFAST identifier prefix, 184, 413
PRIidFAST32 macro, 184, 512
PRIidFAST64 macro, 512
PRIidLEAST identifier prefix, 184, 413
PRIidLEAST32 macro, 512
PRIidLEAST64 macro, 512
PRIidMAX macro, 184, 413, 512
PRIidPTR macro, 184, 413, 512
PRIi identifier prefix, 184, 413
PRIi32 macro, 512
PRIi64 macro, 512
PRIiFAST identifier prefix, 184, 413
PRIiFAST32 macro, 512
PRIiFAST64 macro, 512
PRIiLEAST identifier prefix, 184, 413
PRIiLEAST32 macro, 512
PRIiLEAST64 macro, 512
PRIiMAX macro, 184, 413, 512
PRIiPTR macro, 184, 413, 512
primary expression, 65
printf_s function, 428, 519, 529, 530
printing character, 19, 166, 167
printing wide character, 386
PRIo identifier prefix, 184, 413
PRIo32 macro, 512
PRIo64 macro, 512
PRIoFAST identifier prefix, 184, 413
PRIoFAST32 macro, 512
PRIoFAST64 macro, 512
PRIoLEAST identifier prefix, 184, 413
PRIoLEAST32 macro, 512
PRIoLEAST64 macro, 512
PRIoMAX macro, 184, 413, 512
PRIoPTR macro, 184, 413, 512
primary expression, 65
printf_s function, 428, 519, 529, 530
printing character, 19, 166, 167
printing wide character, 386
PRIo identifier prefix, 184, 413
PRIo32 macro, 512
PRIo64 macro, 512
PRIoFAST identifier prefix, 184, 413
PRIoFAST32 macro, 512
PRIoFAST64 macro, 512
PRIoLEAST identifier prefix, 184, 413
PRIoLEAST32 macro, 512

Index 623
PRIuLEAST64 macro, 512
PRIuMAX macro, 184, 413, 512
PRIuPTR macro, 184, 413, 512
PRIx identifier prefix, 184, 413
PRIx identifier prefix, 184, 413
PRIx2 macro, 512
PRIx64 macro, 512
PRIxFAST identifier prefix, 184, 413
PRIxFAST identifier prefix, 184, 413
PRIxFAST32 macro, 512
PRIxFAST64 macro, 512
PRIxLEAST identifier prefix, 184, 413
PRIxLEAST identifier prefix, 184, 413
PRIxLEAST32 macro, 512
PRIxLEAST64 macro, 512
PRIxMAX macro, 184, 413, 512
PRIxMAX macro, 184, 413
PRIxPTR macro, 184, 413, 512
PRIxPTR macro, 184, 413
program diagnostics, 156
program execution, 11
program file, 9
program image, 10
program name, 11
program name (argv[0]), 11
program parameters, 11
program startup, 10, 11
program structure, 9
program termination, 10, 11, 12
program, conforming, 8
program, strictly conforming, 8
promotions
default argument, 68
integer, 13, 43
prototype, 108, see function prototype
pseudo-random sequence functions, 306
PTRYDIFF_MAX macro, 265, 426, 519
PTRYDIFF_MIN macro, 265, 426, 519
ptrydiff_t type, 76, 77, 249, 255, 260, 265, 277, 284, 355, 360, 426, 492, 519
PTRYDIFF_WIDTH macro, 265, 519
punctuation, 61
putc macro, 268, 294, 427, 519
putchar macro, 268, 294, 427, 519
puts function, 145, 268, 295, 427, 519
putwc function, 268, 369, 433, 519
putwchar function, 268, 369, 433, 519
qsort function, 312, 314, 428, 489, 519
qsort_s function, 429, 519, 539, 540, 541
qualified types, 38
qualified version of type, 38
quantize macro, 31, 176, 330, 519
quantized identifier prefix, 234, 330, 445, 519
quantized128 function, 234, 423, 519
quantized32 function, 234, 423, 519
quantized64 function, 234, 423, 519
quantum, 30
quantum exponent, 30
quantum exponent functions, 234
quantum functions, 234
quantum macro, 31, 330, 519
quantumd identifier prefix, 235, 330, 446, 519
quantumd128 function, 235, 423, 519
quantumd32 function, 235, 423, 519
quantumd64 function, 235, 423, 519
question-mark escape sequence (\?), 58
quick_exit function, 243, 310, 311, 428, 489, 496, 498, 499, 505, 519, 573
quiet NaN, 24
raise function, 242, 243, 244, 250, 309, 424, 496, 519
rand function, 300, 306, 307, 428, 519
RAND_MAX macro, 300, 306, 307, 428, 519
range excess, 25, 45, 130
range error, 199, 205, 206, 208–212, 214, 216–221, 223, 228, 230, 231
rank, see integer conversion rank
read-modify-write operations, 14
read-read coherence, 16
read-write coherence, 16
real floating type conversion, 44, 447
real type domain, 37
real types, 37
real-floating, 199
realloc function, 307, 308, 309, 393, 428, 489, 498, 505, 519
recommended practice, 6
recursion, 68
recursive function call, 68
redefinition of macro, 139
reentrancy, 12, 19
library functions, 155
referenced type, 38
register storage-class specifier, 88, 131
relational expressions, 77
relaxed atomic operations, 14
release fence, 253
release operation, 14
release sequence, 15
reliability of data, interrupted, 12
remainder assignment operator (\%), 83
remainder function, i, 226, 329, 331, 417, 441, 442, 445, 465, 484, 504, 519
remainder functions, 225, 465
remainder operator (%), 40, 75
remainder type-generic macro, 329
remainderd128 function, 226, 422, 519
remainderd32 function, 226, 422, 519
remainderd64 function, 226, 422, 519
remainderf function, 226, 417, 519
remainderl function, 226, 417, 519
remove function, 271, 272, 426, 504, 519, 525
remquo function, 226, 329, 417, 441, 442, 445, 465, 488, 504, 519
remquo type-generic macro, 329
remquof function, 226, 417, 519
remquol function, 226, 417, 519
rename function, 271, 426, 504, 519
representation
  canonical, 24
  non-canonical, 24
representations of types, 39
pointer, 38
rescanning and replacement, 141
reserved identifier, 508, 514
reserved identifiers, 49, 153, 523
restartable multibyte/wide character conversion functions, 350, 382, 566
restartable multibyte/wide string conversion functions, 383, 567
restore calling environment function, 240
restrict type qualifier, 97, 100
restrict-qualified type, 38, 98
rewind function, 274, 295, 298, 369, 427, 519
right-shift assignment operator (>>=), 83
right-shift operator (>>), 40, 77
rint function, 222, 329, 331, 417, 441, 442, 447, 463, 464, 484, 519
rint type-generic macro, 329
rintd128 function, 222, 421, 519
rintd32 function, 222, 421, 519
rintd64 function, 222, 421, 519
rintf function, 222, 417, 519
rintl function, 222, 417, 519
rootn function, 218, 219, 416, 446, 461, 519
rootn type-generic macro, 329
rootnd128 function, 219, 421, 519
rootnd32 function, 219, 421, 520
rootnd64 function, 219, 421, 520
rootnf function, 219, 416, 520
rootnl function, 219, 416, 520
round function, 197, 223, 224, 225, 329, 412, 417, 422, 442, 463, 464, 484, 520
round to narrower type, 232
round type-generic macro, 329
roundd128 function, 223, 421, 520
roundd32 function, 223, 421, 520
roundd64 function, 223, 421, 520
roundeven function, 197, 223, 224, 417, 442, 464, 520
roundeven type-generic macro, 329
roundeven128 function, 224, 422, 520
roundeven32 function, 224, 422, 520
roundeven64 function, 224, 422, 520
roundeven1 function, 224, 417, 520
roundf function, 223, 417, 520
rounding, 223
rounding control pragma, 174
rounding directions, 197
rounding mode, floating point, 24
roundl function, 223, 417, 520
RSIZE_MAX macro, 426, 520, 524, 525, 526, 530, 531, 534–536, 538, 540–549, 551, 552, 554, 555, 557, 558, 561–564, 566–568
rsqrt function, 219, 416, 446, 462, 520
rsqrt type-generic macro, 329
rsqrttd128 function, 219, 421, 520
rsqrttd32 function, 219, 421, 520
rsqrttd64 function, 219, 421, 520
rsqrtf function, 219, 416, 520
rsqrtl function, 219, 416, 520
runtime-constraint, 6
Runtime-constraint handling functions, 537
rvalue, 45
same scope, 33
samequantum macro, 330, 520
samequantumid identifier prefix, 234, 235, 330, 445, 520
samequantumid128 function, 234, 423, 520
samequantumid32 function, 234, 423, 520
samequantumid64 function, 234, 423, 520
save calling environment function, 240
scalar types, 38
scalbln function, 216, 329, 416, 442, 459, 483, 520
scalbln type-generic macro, 329
scalbld128 function, 216, 420, 520
scalbld32 function, 216, 420, 520
scalbld64 function, 216, 420, 520
scalblnf function, 216, 416, 483, 520
scalblnl function, 216, 416, 483, 520
scalbn function, 216, 329, 416, 442, 457, 458, 459, 474, 483, 520
scalbn type-generic macro, 329
scalbld128 function, 216, 420, 520
scalbld32 function, 216, 420, 520
scalbld64 function, 216, 420, 520
scalblnf function, 216, 416, 483, 520
scalblnl function, 216, 416, 483, 520
scant function, 31, 175, 176, 268, 288, 289, 291, 427, 443, 520, 573
scant function, 428, 520, 530, 534
scanlist, 258, 361
scanset, 285, 361
SCHAR_MAX macro, 21, 23, 413, 438, 520
SCHAR_MIN macro, 21, 23, 37, 413, 438, 520
SCHAR_WIDTH macro, 21, 520
SCN identifier prefix, 184, 392, 508
**Index**

- self-referential structure, 96
- selection statements, 125
- SEEK_SET macro, 268
- setbuf function, 267, 270, 271, 273, 275, 427, 520
- setjmp function, 153, 240, 241, 424, 488, 495, 520, 571
- setlocale function, 152, 189, 190, 193, 345, 413, 495, 503, 520
- setpayload function, 424, 445, 469, 520
- setpayloadd128 function, 424, 469, 520
- setpayloadd32 function, 424, 469, 520
- setpayloadd64 function, 424, 469, 520
- setpayloadf function, 424, 469, 520
- setpayloadl function, 424, 469, 520
- setpayloadsig function, 424, 445, 469, 520
- setpayloadsigd128 function, 424, 469, 520
- setpayloadsigd32 function, 424, 469, 520
- setpayloadsigd64 function, 424, 469, 520
- setpayloadsigf function, 424, 469, 520
- setpayloadsigl function, 424, 469, 520
- setvbuf function, 267, 270, 271, 273, 275, 427, 497, 520
- shall, 8
- shift expressions, 77
- shift sequence, 152
- shift states, 18
- short identifier, character, 20, 51
- short int type, 36, 89
- short int type conversion, 43, 44
- SHRT_MAX macro, 22, 413, 438, 520
- SHRT_MIN macro, 21, 413, 438, 520
- SHRT_WIDTH macro, 22, 520
- side effects, 12, 39, 46, 64, 70, 82, 114, 124, 171, 174, 293, 294, 368, 369, 448, 450, 452
- SIG identifier prefix, 242, 393, 508
- SIG_IGN identifier prefix, 242, 393, 508
- SIG_ATOMIC_MAX macro, 265, 266, 426, 512
- SIG_ATOMIC_MIN macro, 265, 266, 426, 512
- sig_atomic_t type, 12, 242, 243, 265, 266, 424, 487, 496, 520
- SIG_ATOMIC_WIDTH macro, 265, 426, 512
- SIG_DFL macro, 242, 424, 504, 512
- SIG_ERR macro, 242, 424, 496, 512
- SIG_IGN macro, 242, 424, 500, 512
- SIGABRT macro, 242, 309, 424, 512
- SIGFPE macro, 198, 242, 243, 424, 485, 496, 500, 508, 512
- SIGILL macro, 242, 424, 496, 500, 512
- SINGINT macro, 242, 424, 512
- sign and magnitude, 40
- sign bit, 40
- signal function, 13, 14, 112, 242, 243, 310, 311, 424, 496, 500, 504, 520
- signal handler, 12, 19, 242, 244
- signal handling functions, 242
- signal handling header, 242
- signaling NaN macros, 196
- signaling NaN, 24, 441
- signals, 12, 19, 242
- signbit macro, 201, 414, 443, 465, 520

- SCNc:FASTN macros, 184
- SCNc:LEASTN macros, 184
- SCNc:MAX macro, 184
- SCNc:N macros, 184
- SCNc:PTR macro, 184
- SCNd identifier prefix, 184, 413
- SCNd:FAST identifier prefix, 184, 413
- SCNd:LEAST identifier prefix, 184, 413
- SCNd:MAX macro, 184, 413, 512
- SCNd:PTR macro, 184, 413, 512
- SCNi identifier prefix, 184, 413
- SCNi:FAST identifier prefix, 184, 413
- SCNi:LEAST identifier prefix, 184, 413
- SCNi:MAX macro, 184, 413, 512
- SCNi:PTR macro, 184, 413, 512
- SCNo identifier prefix, 184, 413
- SCNo:FAST identifier prefix, 184, 413
- SCNo:LEAST identifier prefix, 184, 413
- SCNo:MAX macro, 184, 413, 512
- SCNo:PTR macro, 184, 413, 512
- SCNx identifier prefix, 184, 413
- SCNx:FAST identifier prefix, 184, 413
- SCNx:LEAST identifier prefix, 184, 413
- SCNx:MAX macro, 184, 413, 512
- SCNx:PTR macro, 184, 413, 512
- scope, 33
- scope of identifier, 33, 133
- search functions
  - string, 322, 547
  - utility, 312, 539
  - wide string, 377, 564
- SEEK_CUR macro, 268, 297, 426, 520
- SEEK_END macro, 268, 270, 297, 426, 520
- SEEK_SET macro, 268, 297, 298, 426, 498, 520
- selection statements, 125
- self-referential structure, 96
- semicolon punctuator (;), 87, 90, 124, 127, 128
- separate compilation, 9
- separate translation, 9
- sequence points, 12, 68, 80, 81, 84, 98, 100, 123, 154, 275, 312, 354, 436, 539
- sequenced after, see sequenced before
- sequenced before, 12, 64, 68, 70, 82, see also indeterminately sequenced, un-sequence
- sequencing of statements, 123
- sequential consistency, 17
- set_constrained_handler_s function, 429, 520, 523, 537, 538
- setbuf function, 267, 270, 271, 273, 275, 427, 520
- signbit function, 201, 414, 443, 465, 520
signed char type, 36
signed character, 43
signed integer types, 36, 43, 53
signed type, 89
signed type conversion, 43, 44
signed types, 36
significand part, 54
SIGSEGV macro, 242, 243, 424, 496, 500, 512
SIGTERM macro, 242, 424, 512
simple assignment, 83
simple assignment operator (=), 83
sin function, 69, 204, 328, 331, 414, 446, 454, 481, 520
sin type-generic macro, 328
sin type-generic macro, 481
sind128 function, 204, 419, 520
sind32 function, 204, 419, 520
sind64 function, 204, 419, 520
sinf function, 204, 414, 520
single-byte character, 18
single-byte/wide character conversion functions, 381
single-precision arithmetic, 13
single-quote escape sequence (‘ ’), 58, 59
singularity, 198
sinh function, 208, 209, 328, 415, 447, 456, 481, 520
sinh type-generic macro, 328
sinh type-generic macro, 481
sind128 function, 208, 419, 520
sind32 function, 208, 419, 520
sind64 function, 208, 419, 520
sinf function, 208, 415, 520
sinhl function, 208, 415, 520
sint function, 204, 414, 520
sipi function, 206, 207, 414, 446, 455, 520
sipi type-generic macro, 329
sipid128 function, 206, 419, 520
sipid32 function, 206, 419, 520
sipid64 function, 206, 419, 520
sipif function, 206, 414, 520
sipill function, 206, 414, 520
SIZE_MAX macro, 265, 426, 520, 524
SIZE_WIDTH macro, 265, 426, 520
sizeof operator, 46, 72, 74
sizes of integer types header, 188
SNAN macro, 196, 441, 520
SNAND128 macro, 196, 520
SNAND32 macro, 196, 520
SNAND64 macro, 196, 520
SNANF macro, 196, 441, 520
SNANL macro, 196, 441, 520
snprintf function, 289, 291, 301, 302, 345, 427, 520, 531, 574
snprintf_s function, 428, 520, 530, 531
snwprintf_s function, 434, 520, 554, 555
sorting utility functions, 312, 539
source character set, 9, 17
source file, 9
name, 145, 148
source file inclusion, 138
source lines, 9
source text, 9
space character (’ ’), 9, 17, 48, 166, 168, 387
space format flag, 276, 355
spilling, 13
sprintf function, 289, 292, 427, 520, 531
sprintf_s function, 428, 520, 531
sqrt function, 219, 328, 331, 416, 441, 442, 462, 467, 520
sqrt type-generic macro, 328
sqrtld128 function, 219, 421, 520
sqrtld32 function, 219, 331, 421, 520
sqrtld64 function, 219, 421, 520
sqrtf function, 219, 416, 520
sqrtl function, 219, 416, 520
square root rounded to narrower type, 233
rand function, 306, 307, 428, 520
sscanf function, 287, 289, 292, 427, 520
sscanf_s function, 428, 520, 532, 535
standard attribute, 117
standard error stream, 268, 270, 299
standard floating types, 36
standard headers, 8, 152
<assert.h>, 152, 153, 156, 180, 410
<ctype.h>, 152, 166, 167, 168, 392, 411
<errno.h>, 152, 170, 392, 411, 523
<fpclassify.h>, 8, 12, 14, 24, 29, 32, 84, 152, 171, 173–175, 177–182, 198, 392, 411, 442, 444, 448–450, 459, 463–465, 485, 574
<float.h>, 8, 20, 23, 24, 27, 28, 29, 152, 183, 280, 303, 358, 371, 392, 412, 438, 447, 505, 572, 573
<inttypes.h>, 152, 184, 185, 186, 392, 413, 573
<iso646.h>, 8, 152, 187, 413, 573, 574
<limits.h>, 8, 20, 21, 36, 37, 152, 188, 413, 438, 505
<locale.h>, 153, 189, 190, 392, 413
<setjmp.h>, 153, 240, 241, 424
<signal.h>, 153, 242, 243, 393, 424
<stdalign.h>, 8, 153, 245, 425, 572
<stdarg.h>, 8, 108, 153, 246, 247, 248,
null, 124
return, 129, 448
selection, 125
sequencing, 123
switch, 125
while, 127
static assertions, 117
static storage duration, 35
static storage-class specifier, 34, 35, 88
static, in array declarators, 106, 108
static_assert declaration, 117
static_assert macro, ii, 156, 410, 520
STDC pragma, 84, 146, 151, 158, 173–175, 179,
180, 182, 199, 408, 410, 412, 414, 449,
450, 459, 463–465, 473, 494, 503, 520
stderr stream, 145, 268, 269, 270, 274, 290,
364, 426, 507, 520
stdin stream, 268, 269, 270, 274, 286–288, 294,
363, 366, 368, 369, 426, 520, 529, 530,
536, 560
stdout stream, 268, 269, 270, 274, 281, 282,
288, 294, 295, 359, 366, 369, 426, 520
storage duration, 35
storage order of array, 67
storage unit (bit-field), 39, 92
storage-class specifiers, 88, 151
store and load, 13
str identifier prefix, 392, 393, 508
strcat function, 320, 429, 513
strcat_s function, 430, 513
strchr function, 323, 430, 513
strchr_s function, 321, 322, 429, 513
strcoll function, 190, 321, 322, 429, 513
strcpy function, 287, 320, 429, 513
strcpys function, 430, 513, 545
strcspsn function, 323, 430, 513
strdup function, ii, xiii, 326, 430, 513, 572
streams, 268, 310
fully buffered, 270
line buffered, 270
orientation, 269
standard error, 268, 270
standard input, 268, 270
standard output, 268, 270
unbuffered, 270
strerror function, 299, 325, 430, 499, 506, 513
strerror_s function, 325, 430, 513, 549, 550
strerrorlen_s function, 430, 513, 550
strfrom identifier prefix, 175, 176
strfromd function, 301, 428, 443, 513
strfromd identifier prefix, 301, 302
strfromd128 function, 301, 429, 513
strfromd32 function, 301, 429, 513
strfromd64 function, 301, 429, 513
strfromf function, 301, 428, 513
strfroml function, 301, 428, 513
strftime function, 190, 322, 430, 499, 513
strictly conforming program, 8
string, 152
  comparison functions, 321
  concatenation functions, 320, 546
  conversion functions, 190
  copying functions, 319, 543
  library function conventions, 319
  literal, 10, 17, 46, 59, 65, 113
  miscellaneous functions, 325, 549
  numeric conversion functions, 185
  search functions, 322, 547
string duplicate function, 326
string handling header, 319, 393, 543
stringizing, 140, 149
strlen function, 321, 326, 430, 513
strncat function, 321, 429, 513
strndup function, 430, 513
stronger, 42
strpbrk function, 323, 430, 513
strrchr function, 324, 430, 513
strncpy function, 324, 430, 513
strncpy_s function, 430, 513
strndup function, ii, xii, 326, 430, 513, 572
strnlen function, 513, 545–547, 550
stronger, 42
strtok function, 323, 430, 513
strrchr function, 323, 324, 430, 513
strsbrk function, 324, 430, 513
strrtm function, 324, 430, 513
strtolf function, 324, 388, 301, 302, 428, 489, 505, 513
strtoimax function, 185, 413, 513
strtok function, 324, 325, 430, 499, 513
strtok_s function, 325, 430, 513, 547, 548, 549
strtol function, 185, 283, 284, 288, 301, 305, 306, 428, 513
strtold function, 227, 288, 301, 302, 428, 489, 505, 513
strtoll function, 185, 283, 284, 301, 305, 306, 428, 513
strtoul function, 185, 283, 284, 301, 305, 306, 428, 513
strtoumax function, 185, 413, 513
structure
  arrow operator (->), 69
  content, 96
dot operator (.), 68
initialization, 113
member alignment, 92
member name space, 34
member operator (.), 46, 68
pointer operator (->), 69
specifier, 90
tag, 34, 96
type, 37, 90
structure content, 96
strxfrm function, 190, 322, 430, 499, 513
subnormal floating-point numbers, 24
subscripting, 67
subtract and round to narrower type, 232
subtraction assignment operator (+=), 93
subtraction operator (–), 40, 76, 474
successful termination, 310
suffix
  floating constant, 55
  integer constant, 53
switch body, 125
switch case label, 123, 125
switch default label, 123, 125
switch statement, 124
swprintf function, 363, 365, 432, 520, 554, 555
swprintf_s function, 434, 520, 555
swscanf function, 363, 365, 432, 520
swscanf_s function, 434, 520, 555, 556, 558
symbols, 3
synchronization operation, 14
synchronize with, 15
syntactic categories, 33
syntax notation, 33
syntax rule precedence, 9
syntax summary, language, 395
system function, 311, 312, 428, 499, 500, 505, 520
t format modifier, 277, 284, 355, 360
tab characters, 17, 48
tag compatibility, 40
tag name space, 34
tags, 34, 95
tan function, 204, 328, 414, 446, 454, 481, 520
tan type-generic macro, 328
tan type-generic macro, 481
tand128 function, 204, 419, 520
tand2 function, 204, 419, 520
tand64 function, 204, 419, 520
tanf function, 204, 414, 520
tanh function, 209, 328, 415, 447, 456, 481, 520
tanh type-generic macro, 328
tanh type-generic macro, 481
tanh128 function, 208, 209, 419, 520
thousands_sep function, 208, 209, 419, 520
tanhd32 function, 208, 209, 419, 520
tanhd64 function, 208, 209, 419, 520
tanhf function, 209, 415, 520
tanh1 function, 209, 415, 520
tanl function, 204, 414, 520
tanpi function, 207, 414, 446, 456, 520
tanpi type-generic macro, 329
tanpid128 function, 207, 419, 520
tanpid32 function, 207, 419, 520
tanpid64 function, 207, 419, 520
tanpif function, 207, 414, 520
tanpil function, 207, 414, 520

temporary lifetime, 35
tentative definition, 133

text streams, 268, 295, 297, 298
tgammal function, 221, 329, 416, 462, 520
tgammal type-generic macro, 329
tgammad128 function, 221, 421, 520
tgammad32 function, 221, 421, 520
tgammad64 function, 221, 421, 520
tgammad function, 221, 416, 520
tgammal function, 221, 416, 520

thousands_sep structure member, 189, 191, 520

thrd__ identifier prefix, 394, 508
thrd_busy constant, 333, 337, 431, 513
thrd_create function, 332, 337, 431, 513
thrd_current function, 337, 431, 513
thrd_detach function, 338, 431, 499, 513
thrd_equal function, 338, 431, 513
thrd_error constant, 333, 334–341, 431, 513
thrd_exit function, 337, 338, 431, 489, 513
thrd_join function, 338, 339, 431, 499, 513
thrd_nomem constant, 333, 334, 337, 431, 513
thrd_sleep function, 339, 431, 513
thrd_start_t type, 332, 337, 431, 513
thrd_success constant, 333, 334–341, 431, 513

thrd_t type, 332, 337, 438, 513

thrd_timedout constant, 333, 335, 336, 431, 513

thrd_yield function, 339, 431, 513

thread, 14

thread of execution, 14, 155, 171, 311, 539

thread storage duration, 35, 171

thread_local macro, 332, 431, 520

threads header, 332, 394

time

broken down, 342, 343, 345–347, 551–553

calendar, 342, 343, 344, 346, 552, 553

components, 342, 550

conversion functions, 345, 550

wide character, 380

local, 342

manipulation functions, 343

normalized broken down, 550, 551

time base, 342, 344

time function, 343, 344, 431, 489, 520

TIME__ identifier prefix, 393, 508

TIME_t type, 342, 343, 344, 346, 431, 432, 505, 520, 552

TIME.Utc macro, 335, 336, 339, 342, 434, 431, 505, 513

timespec structure type, 334, 336, 339, 342, 434, 431, 520

timespec_get function, 344, 431, 520


tm_hour structure member, 342, 344, 345, 347, 348, 520, 551

tm_isdst structure member, 343, 344, 348, 520

tm_mday structure member, 342, 343–345, 347, 520, 551

tm_min structure member, 342, 344, 345, 347, 348, 520, 551

tm_mon structure member, 342, 343–345, 347, 520, 551

tm_sec structure member, 342, 344, 345, 347, 348, 521, 551

tm_wday structure member, 343, 344, 345, 347, 348, 521, 551

tm_yday structure member, 343, 347, 348, 521

tm_year structure member, 343, 344, 345, 347, 348, 521, 551

TMP_MAX macro, 268, 272, 426, 520

TMP_MAX_S macro, 427, 520, 524, 525, 526
tmpfile function, 272, 310, 427, 520
tmpfile_s function, 427, 520, 525, 526
tmpnam function, 267, 268, 272, 427, 521, 526
tmpnam_s function, 427, 521, 524, 525, 526
to identifier prefix, 392, 394, 508
token, 10, 48, see also preprocessing tokens

token concatenation, 140
token pasting, 140
tolower function, 168, 411, 513
totalorder function, i, 424, 444, 467, 468, 513
totalorder128 function, 424, 468, 513
totalorder32 function, 424, 468, 513
totalorder64 function, 424, 468, 513
totalorderf function, 424, 467, 513
totalorderl function, 424, 468, 513
totalordermag function, 424, 444, 468, 513
totalordermagd128 function, 424, 468, 513
totalordermagd32 function, 424, 468, 513
totalordermagd64 function, 424, 468, 513
totalordermagf function, 424, 468, 513
totalordermagl function, 424, 468, 513
toupper function, 168, 169, 411, 513
towctrans function, 390, 391, 435, 500, 506, 513
towlower function, 390, 391, 435, 513
towupper function, 390, 391, 435, 513
translation environment, 9
translation limits, 19
translation phases, 9
translation unit, 9, 131
trap, see perform a trap
trap representation, 7, 39, 40, 46, 68
trigonometric functions
  complex, 158, 475
  real, 202, 454
trigraph sequences, 9, 18
true macro, 235, 259, 393, 426, 452, 482, 521
trunc function, 197, 224, 329, 417, 442, 464, 484, 521
  type-generic macro, 329
truncation, 44, 224, 270, 273
truncation toward zero, 75
truncd128 function, 224, 422, 521
truncd32 function, 224, 422, 521
truncd64 function, 224, 422, 521
truncf function, 224, 417, 521
truncl function, 224, 417, 521
tss_identifier prefix, 394, 508
tss_create function, 339, 340, 431, 499, 500, 513
tss_delete function, 340, 431, 489, 500, 513
tss_dtor_t type, 332, 339, 431, 513
tss_get function, 340, 431, 500, 513
tss_set function, 340, 341, 431, 500, 513
  tv_nsec structure member, 342, 344, 521
tv_nsec structure member, 342, 344, 521
two’s complement, 40, 261
type, 36
  type category, 38
  type conversion, 42
  type definitions, 111
  type domain, 37, 471
  type name, 110
  type names, 110
  type punning, 68
  type qualifiers, 97
  type specifiers, 89
  type-generic macro, 481
  type-generic macros, 327
type-generic math header, 327
typedef declaration, 111
typedef storage-class specifier, 88, 111
types, 36
  atomic, 12, 38, 39, 46, 69, 70, 83, 97, 149, 254
  character, 113
  compatible, 40, 90, 98, 105
  complex, 37, 471
  composite, 41
  const qualified, 98
  conversions, 42
  imaginary, 471
  restrict qualified, 98
  volatile qualified, 98
U encoding prefix, 57–60, 398
u encoding prefix, 57–60, 398
u8 encoding prefix, 57–59, 398
UCHAR_MAX macro, 21, 23, 413, 438, 521
UCHAR_WIDTH macro, 21, 521
ufromfp function, 197, 224, 225, 417, 443, 447, 464, 521
ufromfp functions, 224
ufromfp type-generic macro, 329
ufromfpd128 function, 224, 422, 521
ufromfpd32 function, 224, 422, 521
ufromfpd64 function, 224, 422, 521
ufromfpf function, 224, 417, 521
ufromfpl function, 224, 417, 521
ufromfpx function, 197, 225, 417, 443, 447, 464, 521
ufromfpx functions, 225
ufromfpx type-generic macro, 329
ufromfpxd128 function, 225, 422, 521
ufromfpxd32 function, 225, 422, 521
ufromfpxd64 function, 225, 422, 521
ufromfpfx function, 225, 417, 521
ufromfpfxl function, 225, 417, 521
UINT identifier prefix, 263, 266, 393, 426, 508
uint identifier prefix, 261, 393, 426, 508
UINTN_C macros, 266
UINTN_MAX macros, 263
uintN_t types, 261
UINTN_WIDTH macros, 263
UINT16_C macro, 513
UINT16_MAX macro, 513
uint16_t type, 513
UINT32_C macro, 513
UINT32_MAX macro, 513
uint32_t type, 513
UINT64_C macro, 266, 513
UINT64_MAX macro, 513
uint64_t type, 225, 513
UINT8_C macro, 513
UINT8_MAX macro, 513
uint8_t type, 513
UINT_FAST identifier prefix, 264, 426
uint_fast identifier prefix, 262, 426
UINT_LEAST identifier prefix, 263, 426
uint_least identifier prefix, 261, 266, 426
UINT_FASTN_MAX macros, 264
UINT_FASTN_WIDTH macros, 264
uint_fast16_t type, 255, 262, 513
uint_fast32_t type, 255, 262, 513
uint_fast64_t type, 255, 262, 513
uint_fast8_t type, 255, 262, 513
uint_fastN_t types, 262
UINT_LEASTN_MAX macros, 263
uint_leastN_t types, 261
UINT_LEASTN_WIDTH macros, 263
uint_least16_t type, 255, 261, 262, 350, 513
uint_least32_t type, 255, 262, 350, 513
uint_least64_t type, 255, 262, 266, 513
uint_least8_t type, 254, 262, 513
UINT_MAX macro, 22, 137, 413, 438, 482, 514
UINT_WIDTH macro, 22, 514
UINTMAX_C macro, 266, 426, 514
UINTMAX_MAX macro, 184–186, 265, 426, 447, 514
uintmax_t type, 137, 184–186, 224, 225, 255, 262, 266, 277, 284, 355, 360, 413, 417, 422, 426, 514, 573
UINTMAX_WIDTH macro, 265, 426, 514
UINTPTR_MAX macro, 264, 426, 514
uintptr_t type, 255, 262, 426, 514
UINTPTR_WIDTH macro, 264, 426, 514
ULLONG_MAX macro, 23, 306, 374, 413, 438, 482, 521
ULLONG_WIDTH macro, 23, 521
ULONG_MAX macro, 22, 306, 374, 413, 438, 482, 521
ULONG_WIDTH macro, 22, 521
unary arithmetic operators, 73
unary expression, 72
unary minus operator (-), 73, 444
unary operators, 72
unary plus operator (+), 73
unbuffered, 270
unbuffered stream, 270
undef, 135, 136, 142, 147, 154, 155, 407, 494, 521
undef preprocessing directive, 142, 154
undefined behavior, 4, 8, 490
underscore character, 50
underscore, leading, in identifier, 153
ungetc function, 268, 295, 297, 393, 427, 488, 498, 508, 521, 574
ungetwc function, 268, 369, 370, 433, 488, 508, 521
Unicode, 350, see also char16_t type, char32_t type, wchar_t type
Unicode required set, 148
unicode utilities header, 350
union
arrow operator (->), 69
content, 96
dot operator (.), 68
initialization, 113
member alignment, 92
member name space, 34
member operator (.), 46, 68
pointer operator (->), 69
specifier, 90
tag, 34, 96
type, 37, 90
union content, 96
universal character name, 51
unnormalized floating-point numbers, 24
unqualified type, 38
unqualified version of type, 38
unsequenced, 12, 64, 82, see also indeterminately sequenced, sequenced before
unsigned integer suffix, u or U, 53
unsigned integer types, 36, 43, 53
unsigned type, 89, 277, 284, 355, 360
unsigned type conversion, 43, 44
unsigned types, 36
unspeicified behavior, 4, 8, 487
unspecified value, 6
unsuccessful termination, 309, 310
uppercase letters, 17
use of library functions, 154
USHRT_MAX macro, 22, 413, 438, 521
USHRT_WIDTH macro, 22, 521
usual arithmetic conversions, 44, 45, 75, 76, 78–81
UTF–16, 148
UTF–32, 148
UTF–8 string literal, see string literal
utilities, general, 300, 393, 537
wide string, 370, 560
utilities, unicode, 350
va_copy function, 153, 246, 247, 248, 425, 488, 496, 521, 574
value, 6
value bits, 39
value of a string, 152
value of a wide string, 152
variable arguments, 140
variable arguments header, 246
variable length array, 105, 106, 149
variably modified, 105
variably modified type, 105, 106, 149
vertical tab, 19
vertical-tab character, 17, 48
vertical-tab escape sequence (\v), 19, 58, 168
vfprintf function, 268, 290, 427, 498, 521, 532
vfprintf_s function, 428, 521, 532, 533–535
vfscanf function, 268, 290, 427, 498, 521
vfscanf_s function, 428, 521, 533, 534, 535
vfwprintf function, 268, 364, 432, 498, 521, 556
vfwprintf_s function, 434, 521, 556
vfwscanf function, 268, 364, 365, 369, 432, 498, 521
vfwscanf_s function, 434, 521, 556, 557–559
visibility of identifier, 33
visible, 33
visible side effect, 16
VLA, see variable length array
void expression, 46
void function parameter, 108
void type, 46, 89
void type conversion, 46
volatile access, 11, 12
volatile type qualifier, 97
volatile-qualified type, 38, 98
vprintf function, 268, 290, 291, 427, 498, 521, 533
vprintf_s function, 428, 521, 533, 534, 535
vsprintf function, 268, 290, 291, 427, 498, 521, 574
vsprintf_s function, 428, 521, 533, 534–536
vsnprintf function, 290, 291, 427, 498, 521, 534
vsnprintf_s function, 428, 521, 533, 534, 535
vswprintf function, 434, 521, 557, 558
vswprintf_s function, 434, 521, 557, 558
vswscanf function, 434, 521, 557, 558
vswscanf_s function, 434, 521, 557, 558, 559
vswprintf function, 268, 364, 365, 366, 432, 498, 521, 559
vswprintf_s function, 434, 521, 558, 559
vswscanf function, 268, 364, 366, 369, 432, 498, 521
vswscanf_s function, 434, 521, 557, 558, 559

warnings, 10, 486

WCHAR_MAX macro, 265, 266, 353, 426, 432, 521
WCHAR_MIN macro, 265, 266, 353, 426, 432, 521

WCHAR_WIDTH macro, 265, 426, 521
wcrtomb_s function, 434, 521, 566, 567
wcs identifier prefix, 392–394, 508
wcs cat function, 375, 433, 514
wcs cat_s function, 434, 514, 563
wcschr function, 377, 433, 514
wcsc mp function, 375, 376, 433, 514
wscoll function, 376, 433, 514
wcsncpy function, 374, 433, 514
wcsncpy_s function, 434, 514, 561, 562
wcsnlen function, 434, 514, 561, 563, 564, 565, 566
wcs pbrk function, 378, 433, 514
wcs rchr function, 378, 433, 514
wcsrtombs function, 384, 385, 433, 514, 567
wcsrtombs_s function, 434, 514, 567, 568, 569
wcsspn function, 378, 433, 514
wctob function, 31, 32, 175, 176, 178, 372, 514
wcto family, 31, 32, 175, 176, 372, 514
wcto family, 317, 372, 373
wcstol function, 360, 361, 363, 370, 433, 443, 489, 505, 514
wcstol function, 370, 372, 505, 514
wcstod function, 370, 371, 505, 514
wcstod function, 370, 371, 505, 514
wcstod function, 370, 371, 505, 514
wcstof function, 363, 370, 433, 489, 505, 514
wcstoln function, 185, 186, 413, 514
wcstok function, 379, 433, 499, 500, 514
wcstok function, 434, 514, 564, 565
wcstol function, 186, 360, 361, 363, 373, 374, 433, 514
wcst oll function, 363, 370, 433, 489, 505, 514
wcst oll function, 186, 363, 373, 374, 433, 514
wcstombs function, 317, 383, 429, 514
wcstombs_s function, 429, 514, 542, 543
wcstoull function, 186, 361, 363, 373, 374, 433, 514
wcstoull function, 186, 363, 373, 374, 433, 514
wcstoumax function, 185, 186, 413, 514
wcsxf r function, 376, 433, 499, 514
wctob function, 381, 386, 433, 521
wctomb function, 315, 316, 317, 382, 429, 521
wctomb_s function, 429, 521, 541
wc tran s function, 390, 391, 435, 500, 521
wctrans_t type, 386, 390, 391, 434, 435, 521
wctype function, 389, 390, 435, 500, 521
wctype_t type, 386, 389, 390, 434, 435, 521
weaker, 42
WEOF macro, 353, 367–370, 381, 386, 432, 434, 500, 521
white space, 9, 48, 136, 168, 388
white-space character, 152
white-space characters, 48
white-space wide character, 152
wide character, 5
case mapping functions, 390
extensible, 390
classification functions, 386
extensible, 389
constant, 57
formatted input/output functions, 354, 553
input functions, 268
input/output functions, 268, 366
output functions, 268
single-byte conversion functions, 381
wide character classification and mapping utilities header, 386, 394
wide character constant, 57
wide character input functions, 268
wide character input/output functions, 268
wide character output functions, 268
wide string, 152
wide string comparison functions, 375
wide string concatenation functions, 375, 563
wide string copying functions, 374, 560
wide string literal, see string literal
wide string miscellaneous functions, 380, 565
wide string numeric conversion functions, 185, 370
wide string search functions, 377, 564
wide-oriented stream, 269
width, 40
WINT_MAX macro, 265, 266, 426, 521
WINT_MIN macro, 265, 266, 426, 521
wint_t type, 265, 266, 277, 279, 281, 353, 355, 357, 367–369, 381, 386–390, 432–435, 500, 521
WINT_WIDTH macro, 265, 426, 521
wmemchr function, 379, 380, 433, 521
wmemcmp function, 376, 377, 433, 521
wmemcpy function, 374, 433, 521
wmemcpy_s function, 434, 521, 562
wmemmove function, 374, 375, 433, 521
wmemmove_s function, 434, 521, 562, 563
wmemset function, 380, 433, 521
wprintf function, 175, 176, 184, 268, 365, 366, 432, 443, 521, 560
wprintf_s function, 434, 521, 559, 560
write-read coherence, 16
write-write coherence, 16
wscanf function, 175, 176, 268, 366, 369, 432, 443, 521
wscanf_s function, 434, 521, 559, 560
xor macro, 187, 413, 521
xor_eq macro, 187, 413, 521
z format modifier, 277, 284, 355, 360
zero, 471